

Laboratoire d'Analyse
Numérique et Informatique

Université de Franche-Comté
Faculté des sciences et des techniques

Besançon

THÈSE

Soutenue le 23 juin 1980 pour obtenir le

**DIPLÔME DE DOCTEUR EN MÉTHODES D'APPROXIMATION
ET ALGORITHMES EN ANALYSE ET THÉORIE DES NOMBRES**

par

El maati CHABBAR

★

1980

**CONTRÔLE ET GESTION DU PARALLÉLISME
TRIS SYNCHRONES ET ASYNCHRONES**

Devant la Commission d'examen :

MM. LHOTE F.

LAFON J.C.
STAMON G.
TREHEL M.

Président

}
Examineurs

Librairie de la Sorbonne
Bibliothèque de la Sorbonne

Librairie de la Sorbonne
Bibliothèque de la Sorbonne

THÈSE

Présentée le 20 mai 1970 par M. J. L.

DIPLOME DE DOCTEUR EN METHODES D'APPROXIMATION
ET ALGORITHMES EN ANALYSE ET THEORIE DES NOMBRES

Par M. J. L.

CONTRÔLE ET GESTION DU PARALLÉLISME
TRIS SYNCHRONES ET ASYNCHRONES

Présentée à l'Université de Paris

MAI 1970

MAI 1970

MAI 1970
MAI 1970
MAI 1970

MAI 1970



Digitized by the Internet Archive
in 2026

https://archive.org/details/IA41546609_0018

Laboratoire d'Analyse
Numérique et Informatique

Université de Franche-Comté
Faculté des sciences et des techniques

THÈSE

Soutenue le 23 juin 1980 pour obtenir le

**DIPLÔME DE DOCTEUR EN MÉTHODES D'APPROXIMATION
ET ALGORITHMES EN ANALYSE ET THÉORIE DES NOMBRES**

par

El maati CHABBAR

★

**CONTRÔLE ET GESTION DU PARALLÉLISME
TRIS SYNCHRONES ET ASYNCHRONES**

★★★

Devant la Commission d'examen :

MM. LHOTE F.

Président

LAFON J.C.
STAMON G.
TREHEL M.

}

Examineurs

THÈSE

Présentée à la Faculté des Sciences de la Sorbonne

DIPLOME DE DOCTEUR EN METHODES D'APPROXIMATION
T ALGORITHMES EN ANALYSE ET THEORIE DES NOMBRES

Etienne CHABAR



CONTRÔLE ET GESTION DU PARALLÉLISME
TRIS SYNCHRONES ET ASYNCHRONES

UNIV. LEIDEN
STATION II
VERBODEN

UNIVERSITÉ DE FRANCHE-COMTÉ BESANÇON

Président : Monsieur le Professeur Pierre LÉVÊQUE

Président honoraire : Monsieur le Professeur Jean THIÉBAUT

FACULTÉ DES SCIENCES ET DES TECHNIQUES

DOYEN

M. le Professeur Jean BULABOIS
Physique Générale-optique

ASSEESSEURS

M. le Professeur Bernard MILLET
Botanique

M. Robert GUYETANT, Maître-Assistant
Biol. animale Ecologie

DOYENS HONORAIRES

MM. GLANGEAUD, JACQUEMAIN, CHATELET,
THIEBAUT, BERNARD

PROFESSEURS HONORAIRES

MM. BENNETON, CHATELET, MESNAGE, QUANTIN
THEOBALD N., THIRY, TRILLAT, TRONCHET,
UEBERSFELD

MM. BANTEGNIE Robert	: Mathématiques
BENILAN Philippe	: Mathématiques
BERNARD Jean	: Chimie Physique
BIDAULT Michel	: Taxonomie expérimentale
BROQUET Paul	: Géologie
BRUCKERT Sylvain	: Ecologie végétale
BULABOIS Jean	: Physique Gle optique
CAPODANNO Pierre	: Mécanique théorique
CERUTTI Ernest	: Chimie appliquée
CHALEAT Raymond	: Mécanique appliquée
CHARDON J. Claude	: Spectroscopie
CHAUVE Pierre	: Géologie minéralogie
COUGNARD Jean	: Mathématiques
DEVIN Claude	: Chimie 1er cycle
DUBOUCHET Jacques	: Physiologie végétale
GALATRY Louis	: Physique moléculaire
GAUDEMER Yves	: Biochimie
GOMOT Lucien	: Zoologie-Embryologie
GOUARNE René	: Mathématiques
GRAS Georges	: Mathématiques
GREMILLARD Jean	: Mathématiques
KARCHE J. Paul	: Pétrographie Minéralogie
LALLEMENT Gérard	: Mécanique appliquée
LAUDE Bernard	: Chimie organique
LESAINT Pierre	: Mathématiques I
LHOTE François	: Automatique
MERIGOUX Henri	: Cristallographie et Synthèses minérales

MIELLOU J. Claude	: Analyse numérique
MILLET Bernard	: Botanique
MONTAGNER Hubert	: Psychophysiologie
MOREELS Guy	: Astronomie
OLIVIER Marcel	: Electroniq. quantique
PARIZET Jean	: Mathématiques
PERROT Roger	: Chimie générale
PLUVINAGE Philippe	: Physique théorique
POTIER Robert	: Physique des sols
RANGHEARD Yves	: Géologie
REAL Pierre	: Ecologie animale
RIPPLINGER Jean	: Physiologie animale
ROBERT Daniel	: Physique
ROBERT Guy	: Electrochimie
ROBERT Jacques	: Mathématiques I
THEOBALD J. Gérard	: Spectroscopie hertzienne
THIEBAUT Jean	: Pétrogr. minéralogie
TREHEL Michel	: Informatique
VERNEAUX Jean	: Zoologie-Embryologie
VIENOT J. Charles	: Physique Gle optique
WEIL Michel	: Mathématiques

Responsable des services administratifs :

M. Jean-Marie RUNGE

Je tiens à remercier Monsieur LHOTE F., Professeur à l'Université de BESANCON, pour l'honneur qu'il me fait en acceptant de présider le jury.

Je remercie Monsieur le Professeur TREHEL, qui est à l'origine de cette thèse, pour m'avoir dirigé efficacement grâce à ses conseils et ses encouragements. Qu'il veuille bien trouver ici l'expression de toute ma reconnaissance et de ma profonde gratitude.

Je remercie également :

Monsieur LAFON J.C., Professeur d'informatique à l'Université de STRASBOURG et Monsieur STAMON G., Professeur d'informatique à l'I.U.T. de BELFORT, pour l'intérêt qu'ils ont manifesté envers ce travail et pour avoir accepté de participer au jury.

Mes remerciements vont aussi à tous les membres du laboratoire d'analyse numérique et informatique de BESANCON pour le soutien moral qu'ils m'ont apporté et la bonne ambiance dans laquelle ils m'ont permis de travailler.

Enfin, je remercie tout particulièrement Mademoiselle B. MOUCHOTTE pour la lourde tâche dactylographique dont elle s'est acquittée avec patience.

T A B L E D E S M A T I E R E S

TABLE DES MATIERES

INTRODUCTION	1
CHAPITRE I - PARALLELISME ET SYSTEMES PARALLELES	3
1.1. Détection du parallélisme	3
1.2. Systèmes parallèles	4
1.2.1. Classement par séquençement	4
1.2.2. Classement par contrôle	5
1.2.3. Classement par communication entre processeurs	6
1.2.4. Classement par spécialisation ou banalisation des processeurs	7
1.3. Exemple d'architecture multiprocesseur MIMD : MICRAL M	8
CHAPITRE II - CONTROLE ET GESTION DU PARALLELISME	9
A. - Gestion des processus	9
A.1. Parallélisme disjoint	9
A.2. Parallélisme itératif	11
B. - Contrôle des processus	13
B.1. Sémaphores	14
B.2. Sections critiques et S.C. contionnelles	17
B.3. Moniteurs	19
B.3.1. Implémentation de HOARE	20
B.3.2. Implémentation de KESSELS	21
B.4. Module de contrôle	23
B.5. Expressions de chemin	24
B.6. Serializer	27
B.7. Réseaux de Petri	29

CHAPITRE III - PROBLEMES BINAIRES	36
3.1. Présentation	36
3.2. Programmation séquentielle	37
3.3. Exemple de programmation sur ILLIAC IV	38
3.4. Analyse de ces problèmes pour des machines MIMD	40
3.4.1. Gestion des processus	41
3.4.2. Gestion des tâches	44
3.5. Exemple d'application : tri par interclasse- ment naturel	45
CHAPITRE IV - TRIS SYNCHRONES	51
4.1. Introduction	51
4.2. Tri à distance	51
4.3. Tri par interclassement	53
4.3.1. Interclassement "pair-impair".....	53
4.3.2. Interclassement "bitonic".....	60
4.4. Tri par transposition	65
4.5. Exemple de tri sur machine SIMD	66
4.5.1. Interconnexion linéaire	66
4.5.2. Interconnexion bidimensionnelle	68
CHAPITRE V - TRIS ASYNCHRONES	72
5.1. Introduction	72
5.2. Preuve de programmes	73
5.2.1. Axiomes de preuve de programmes séquentiels	73
5.2.2. Preuve de programmes parallèles	74
5.3. Tri par interclassement en parallèle	75
5.3.1. Principe de l'algorithme	75
5.3.2. Programme	77
5.3.3. Commentaires sur le programme	78
5.3.4. Preuve du programme	80

5.4. Estimation de la complexité de l'algorithme par interclassement	91
5.5. Application de cet algorithme au tri par sélection quadratique	95
5.5.1. Principe séquentiel de la sélection quadratique	95
5.5.2. Principe de la parallélisation	95
5.6. Tri par comptage en parallèle	97
5.6.1. Principe séquentiel	97
5.6.2. Parallélisation de cet algorithme .	97
5.6.3. Exemple	99
5.6.4. Programmation parallèle de l'algorithme	101
5.6.5. Complexité	102
5.7. Autres algorithmes de tri en parallèle ...	103
5.7.1. Sélection	103
5.7.2. Quicksort	103
5.7.3. Tri externe "par interclassement"..	103
BIBLIOGRAPHIE	104

I N T R O D U C T I O N

INTRODUCTION

L'idée initiale de cette thèse était d'étudier les outils de synchronisation dont on pouvait avoir besoin dans la programmation de différents algorithmes. Après quoi, nous envisagions de les implanter sur un matériel multiprocesseur.

En fait, nous n'avons pas pu acquérir un tel matériel. L'implantation devenait donc difficile. C'est pourquoi, après avoir fait un panorama de la synchronisation, nous avons fait l'étude, plus poussée que nous le prévoyions au départ, de différents algorithmes parallèles. Les algorithmes asynchrones ont surtout retenu notre attention.

Ce travail se présente comme suit :

Le premier chapitre rappelle les grands principes des systèmes parallèles et décrit quelques architectures.

Le second chapitre détaille les techniques de synchronisation, tant classique comme P et V que modernes comme les expressions de chemin et le "serializer".

La résolution de plusieurs problèmes peut se faire en parallèle de manière arborescente. Nous avons appelé binaires de tels problèmes. Ils se prêtent à une programmation spécifique que nous avons décrite avec les outils traditionnels mais aussi en introduisant des primitives adaptées. Cette étude fait l'objet du troisième chapitre.

Plusieurs algorithmes de tri ont donné lieu indépendamment à parallélisation. Il était alors intéressant de présenter une étude sinon exhaustive, du moins plus globale.

Dans cet esprit, nos deux derniers chapitres s'attachent respectivement aux tris synchrones et aux tris asynchrones. Pour chaque exemple étudié, nous avons présenté les algorithmes connus, les modifications proposées et la programmation.

CHAPITRE I

PARALLELISME ET SYSTEMES PARALLELES

CHAPITRE I

PARALLELISME ET SYSTEMES PARALLELES

De nombreux comportements humains sont de nature parallèle alors que l'ordinateur nous a habitué à raisonner séquentiellement. Pourtant, il y a, dans la programmation, tout un aspect sous-jacent, naturellement parallèle, entre :

- les séquences d'instruction d'un programme
- les instructions d'une même séquence
- les phases de préparation et d'exécution des instructions
- les opérations binaires au cours de l'exécution d'une instruction

et à chacune d'elles correspondent plusieurs niveaux de détection du parallélisme (au niveau de la compilation, de l'exécution, etc...)

1.1. Détection du parallélisme

Un premier procédé permettant de détecter les tâches indépendantes dans un programme séquentiel a été proposé par BERNSTEIN [3] : il permet de déterminer les portions de programme qui peuvent être exécutées en parallèle.

Un programme séquentiel peut être considéré comme une suite d'actions $A_1, A_2, \dots, A_n$ agissant sur un ensemble d'objets O , on distingue un sous-ensemble d'objets modifiés OM_i et un sous-ensemble d'objets lus seulement OL_i : $(O_i = OM_i \cup OL_i)$.

Deux actions A_i et A_j sont dites indépendantes ou parallèles si elles vérifient les conditions suivantes :

$$OL_i \cap OM_j = \emptyset$$

$$OM_i \cap OL_j = \emptyset$$

$$OM_i \cap OM_j = \emptyset$$

C'est à dire si elles peuvent s'exécuter simultanément sur deux machines asynchrones en produisant toujours le même résultat. A partir de ces conditions, un analyseur a été défini pour détecter les tâches indépendantes dans un programme FORTRAN [16] ; cet analyseur permet de déterminer les instants désirables (au plus tôt, au plus tard) du lancement des tâches ainsi détectées sur des processeurs.

1.2. Systèmes parallèles

On présente dans MAZARE [27] une excellente classification des différents types d'architecture. On distingue quatre critères de classement[36].

1.2.1 Classement par séquençement

A) Séquençement par instructions :

Chaque processeur exécute les instructions d'un programme dans l'ordre où elles sont écrites par le programmeur. Ces instructions mettent en jeu un ensemble de données. Selon les flots de ces instructions et de ces données, on distingue (FLYNN[14]) quatre types de machines :

a) "single instruction single data" (SISD) : ce sont les machines classiques qui déroulent séquentiellement un flot d'instructions sur un flot de données.

b) "Single instruction multiple data" (SIMD) : où chaque processeur exécute une même instruction, au même moment, sur des données différentes ; ce type de machine est relativement spécialisé dans le traitement des opérations vectorielles ou ensemblistes : c'est le cas d'ILLIAC IV.

c) "Multiple instruction single data" (MISD) : font partie de cette classe les machines "pipeline" qui exécutent plusieurs instructions à la fois sur un programme ne traitant qu'un flot de données.

d) "Multiple instruction multiple data" (MIMD) : ce sont les multiprocesseurs les plus généraux ; les processeurs exécutent des instructions différentes qui agissent sur des données diverses.

B) Séquençement par les données

Ceci est beaucoup moins classique ; l'ordre d'écriture des instructions n'est plus important. Les machines cadencées par les données utilisent le principe suivant : un programme est représenté par un ensemble d'opérations qui utilisent des données en entrée et produisent ou modifient des données en sortie ; toutes les opérations qui ont de entrées produites peuvent s'exécuter simultanément. Cas du système LAU[8] (language d'assignation unique).

1.2.2. Classement par contrôle

Ce critère est lié à la coopération des processeurs et à la façon dont le travail est réparti entre eux.

a) Contrôle hiérarchique :

Un processeur particulier distribue le travail aux autres, leur alloue

les ressources ou les met en attente : c'est le cas du SIMD.

b) Contrôle coopératif :

Chaque processeur peut donner du travail aux autres ; la gestion des processeurs dépend du hardware.

1.2.3. Classement par communication entre processeurs

Les processeurs communiquent soit entre eux, soit avec des ressources communes. On distingue alors trois cas :

1°) Cas d'une mémoire centrale :

On peut utiliser plusieurs techniques pour connecter les processeurs :

a) bus unique : entraîne un ralentissement considérable et limite donc le nombre de processeurs.

b) multibus : chaque processeur est relié à un bus, un banc mémoire reçoit tous les bus.

c) cross-bar : permet de connecter N processeurs à P bancs mémoires, via une matrice d'aiguillage $N \times P$, ce qui est intéressant dans le cas où N et P sont faibles.

Indépendamment de la mémoire centrale, les processeurs peuvent disposer des mémoires locales qui, suivant le cas, sont visibles au programmeur ou transparentes (caches).

2°) Cas d'une mémoire commune dispersée :

Les processeurs ont leur mémoire locale, ils peuvent adresser des mémoires locales à d'autres processeurs par un système de transformation d'adresses.

3°) Multiprocesseurs sans mémoire commune :

La connexion des processeurs peut être, soit directe en utilisant des bus ou des lignes spéciales, soit par un outil de communication très général.

1.2.4. Classement par spécialisation ou banalisation des processeurs

a) Processeurs spécialisés dès la définition de l'architecture matérielle du système où chacun remplit une fonction bien précise.

b) Processeurs banalisés où chacun est capable de remplir toutes les fonctions du système (excepté la fonction de contrôle lorsqu'il y a contrôle hiérarchique). On admet toujours l'existence d'un processeur spécialisé dans des entrées-sorties bien particulières.

c) Processeurs banalisés mais spécialisables : on peut, lorsqu'ils sont identiques, spécialiser les processeurs au niveau de la définition du logiciel et des applications en répartissant explicitement les différentes fonctions sur les différents processeurs (cas du MICRAL M).

1.3. Exemple d'architecture multiprocesseur MIMD : MICRAL M

MICRAL M est une machine multiprocesseur , de 2 à 8 processeurs, construite et commercialisée par la Société REE. Les processeurs, connectés à une mémoire commune, peuvent, chacun, adresser jusqu'à 64k (de 0 à n dans sa mémoire locale et de n+1 à 64 dans la mémoire commune) ; leur connexion à la mémoire commune est réalisée par un organe d'interface branché sur des bus locaux.

Les processeurs sont banalisés. Ils peuvent exécuter un même programme, situé en mémoire commune, sur des données différentes (dans les mémoires locales). Ils peuvent être aussi spécialisés en plaçant les différents programmes dans les mémoires locales ; dans ce cas, ils travaillent indépendamment ; la mémoire commune sert pour les communications.

Les processeurs sont des microprocesseurs Intel 8080 dotés de :

- un système d'interruption
- une horloge
- un dispositif de démarrage automatique
- un dispositif de génération du pluribus d'accès mémoire
- un dispositif de lecture du masque d'interruption
- un dispositif de lecture du temps de l'horloge temps réel
- un compteur pour le télétype de service.

Ils comportent :

- un accumulateur de 8 bits
- six registres
- cinq indicateurs
- une UAL de 8 bits

C H A P I T R E I I

CONTROLE ET GESTION DU PARALLELISME

CHAPITRE II

CONTROLE ET GESTION DU PARALLELISME

Dans ce chapitre, nous étudions deux types de problèmes posés par le parallélisme; ces problèmes étant liés à la gestion des processus et à leur contrôle.

A. Gestion des processus

Elle consiste en la définition, la naissance et la destruction des processus (c.à.d. activation, suspension, destruction des processus); ceci est lié aux différents types de parallélisme :

A.1. Parallélisme disjoint

Lorsqu'il s'agit d'exécuter plusieurs tâches distinctes en parallèle, il faut créer des processus exprimant ces tâches. Une première solution a été donnée par CONWAY [9] qui consiste à définir des primitives (instructions machines), liées à la création des processus; notées FØRK et JOIN :

FØRK A,J,N où A est une adresse

J: adresse d'un compteur

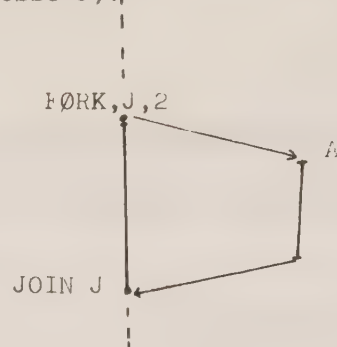
N: nombre de créations de processus.

La rencontre de cette primitive initialise le compteur d'adresse J par la valeur N et provoque l'exécution d'un processus à l'instruction d'adresse A ; celui qui l'exécute continue en séquence.

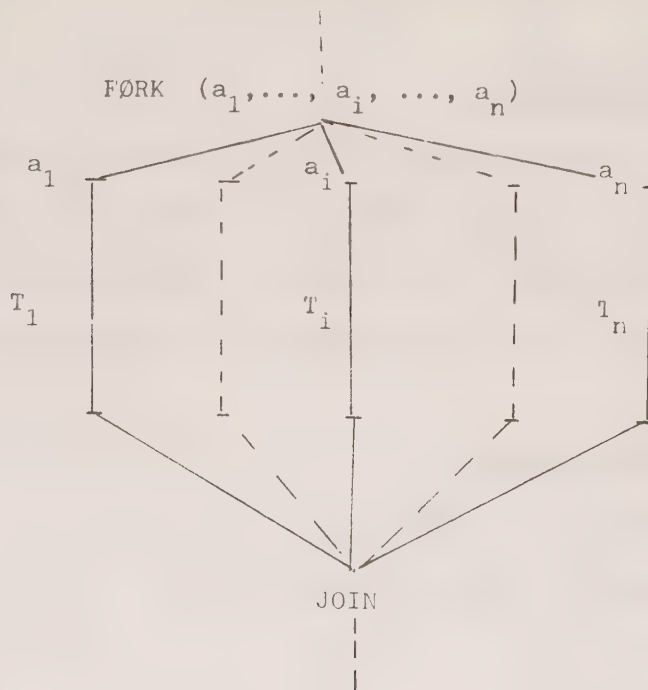
JOIN J : Le compteur d'adresse J est décrementé de 1, en exclusion mutuelle et testé ainsi : si $[J] = 0$ alors aller à J+1

sinon [le processeur exécutant cette primitive disparaît], ($[J]$ signifie contenu de l'adresse J).

Ceci peut être schématisé par : (N=2)



FORK ($a_1, a_2, \dots, a_n$) est une variante de la solution précédente; elle est définie dans le cas où il y a n tâches $(T_i)_{i=1,n}$, d'adresses respectives $(a_i)_{i=1,n}$, à exécuter en parallèle selon le schéma suivant :



Le processeur qui rencontre cette primitive exécute une de ces tâches et lance les autres selon le nombre des processeurs disponibles.

A.2. Parallélisme Itératif : $\left[\begin{smallmatrix} \text{ } \\ 27 \end{smallmatrix} \right]$

Quand il s'agit d'exécuter plusieurs fois une même tâche, il suffit d'activer plusieurs processus en même temps pour faire la même séquence d'instructions. Ce type de parallélisme consiste à traiter les éléments d'un ensemble E et de faire une même action sur chaque élément, jusqu'à ce qu'ils soient tous traités. Ceci peut être schématisé par :

$$\forall x \in E, \text{ faire action sur } x$$

Deux cas peuvent être distingués :

A.2.1. Cas où les éléments de E sont déterminés avant l'exécution.

Deux primitives ont été définies pour exprimer ce type de parallélisme du type NFØRK (a,n), NJOIN.

Remarque : FØRK (a,J,N) conviendrait aussi à ce schéma.

NFØRK permet de lancer des processus en parallèle, selon le nombre de processeurs disponibles, et charge le registre de chaque processeur par le numéro de la boucle d'adresse a : n étant le nombre d'exécutions à faire en parallèle. Après chaque exécution, tout processeur regarde s'il y a des processus non encore démarrés pour relancer l'exécution, ou il sera libéré dans le cas contraire.

A.2.2. Cas où les éléments de E ne sont pas connus à l'exécution.

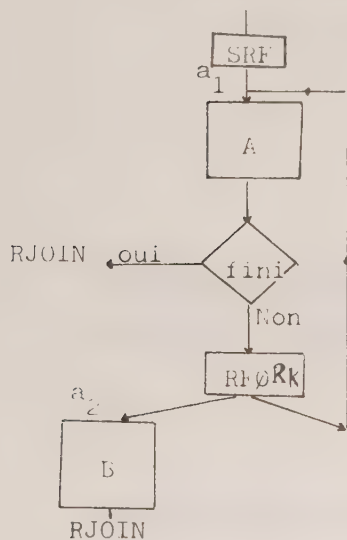
Il est nécessaire dans ce cas, de déterminer d'abord ces éléments et de les traiter au fur et à mesure de leur énumération. On a donc

besoin de définir deux types de processus : processus "énumération" et processus "traitement"; ceci est réalisé par les primitives SRF, RFORK et RJOIN

L'exécution de RFORK (a_1, a_2) active un processus d'énumération à l'adresse a_1 qui pourra rencontrer le RFORK en même temps avec un processus de traitement à l'adresse a_2 ; RJOIN relance les processus d'adresse a_2 non exécutés, libère le processeur, ou continue en séquence suivant les cas.

SRF prépare l'exécution du RFORK qui suivra.

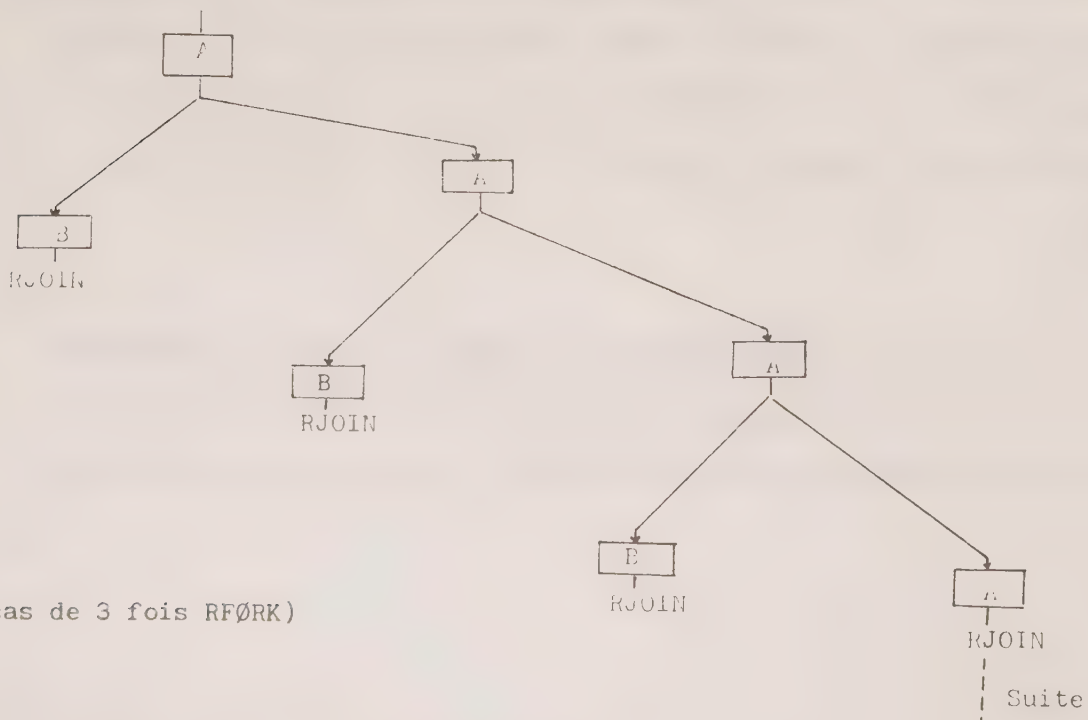
Le programme qui réalise cela a la forme suivante :



où A est le processus d'énumération

B est le processus de traitement

Ce qui donnerait à l'exécution le schéma suivant :



(cas de 3 fois RFORK)

Remarques:

- toutes les exécutions peuvent se faire en parallèle, lancées les unes après les autres.
- il n'y aura jamais deux exécutions simultanées de A.
- dans un système qui ne dispose que d'un FØRK binaire, c'est la seule façon de réaliser N exécutions parallèle d'une séquence, avec N connu à l'avance.

Exemple d'application :

$$E = \left\{ (x,y) \in \mathbb{N}^2 / xy = k \right\}$$

La programmation parallèle du schéma : pour tout $x=(x_1 \ x_2) \in E$ faire action sur x pourrait être implémentée à l'aide de ces mécanismes:

Le processus d'énumération A consiste à chercher les couples (x,y) entiers pour lesquels $x.y=k$ et à chaque couple trouvé, on active un processus (processus B) pour faire une action sur lui.

B. Contrôle des processus

Lorsque les processus coopèrent, il est nécessaire de définir des protocoles permettant de résoudre les problèmes liés à leurs activités simultanées et concurrentes ; on est donc amené à résoudre deux types de problèmes :

- 1°) problème d'accès aux ressources partagées
- 2°) problème de communication entre processus.

Définition CROC [11]:

Une section critique (S.C.) est une phase du processus pendant laquelle une ressource partagée est utilisée et donc inaccessible aux autres processus. Le problème d'accès aux ressources peut être assuré par l'exécution mutuelle.

Pour donner une telle solution, il faut s'assurer que :

- a) à tout instant, un processus au plus peut se trouver en section critique et l'occupe pendant un temps fini.
- b) éviter tout blocage indéfini des processus.
- c) le blocage d'un processus hors d'une section critique ne doit pas empêcher l'entrée d'un autre processus en S.C.

La première idée de la synchronisation était d'utiliser l'indivisibilité d'exécution des instructions machines et de l'exclusion mutuelle, lors de l'accès à une zone partagée en mémoire. Au niveau système, le problème du conflit est assuré par l'utilisation des verrous où les processeurs bouclent sur des instructions spéciales appelées T.A.S. (Test and Set). Avant l'entrée dans sa section critique, on masque le processeur contre toute interruption de façon à ne pas interrompre l'exécution d'autres processus. Ces interruptions seront ensuite démasquées, à la sortie de cette section. L'inconvénient de cette méthode est l'attente active, en ce sens que chaque processus boucle sur l'instruction TAS en monopolisant un processeur.

Nous allons faire un panorama d'outils de synchronisation, en commençant par les primitives de base jusqu'aux outils les plus évolués :

B.1. Sémaphores

Un sémaphore est défini par Dijkstra [12] comme étant un compteur prenant des valeurs entières, auquel est associé une file d'attente (Initialement vide) ; tout sémaphore S est manipulé par deux primitives notées P et V telle que :

P(S) :

début

S: = S-1,

Si $S \leq 0$ alors [le processus qui exécute cette primitive se
bloque et rejoint la file d'attente associée à S] ,

fin

V(S) :

début

S: = S+1

Si $S \leq 0$ alors [un des processus bloqué est oté de la file (asso-
ciée à S) et est remis à l'état actif .]

fin

Le choix du processus de la file d'attente à activer dans V n'est pas lié à la primitive elle-même, mais dépend de la politique de priorité définie pour les files d'attente des sémaphores.

Les deux primitives P et V sont considérées comme étant deux opérations indivisibles, en ce sens que si deux processus tentent d'exécuter simultanément ces deux primitives (ou une même primitive), les deux opérations ne sont exécutées que l'une après l'autre dans un ordre indifférent.

Une autre définition des sémaphores appelée "sémaphores privés", combinée à la précédente, permet de réaliser des solutions de synchronisation plus complexes.

Un sémaphore privé est défini [20] comme étant un sémaphore (initialisé à 0,) associé à un processus particulier qui est le seul à effectuer sur lui des opérations P; tout autre processus ne peut exécuter que des opérations V. Son avantage est de mémoriser un nombre quelconque d'activations, pendant

que le processus auquel elles sont destinées se trouve à l'état actif.

Exemple [20]

Soit A et B deux processus coopérant à une certaine tâche (A processus de calcul par exemple, B processus d'entrée sortie). Au départ, A est actif, B est bloqué. Arrivé à un point de son exécution, A active B pour exécuter sa part de travail. A ce moment A et B se déroulent en même temps. Quand B a terminé, il en avertit A, que celui-ci soit actif ou bloqué.

La solution s'écrit comme suit :

Soit Apriv et Bpriv deux sémaphores privés associés respectivement à A et à B t.q

Apriv = Bpriv = 0

Processus A : début

A_1 ;

V(Bpriv);

A_2 ;

P(Apriv);

A_3 ;

fin

Processus B : début

P (Bpriv);

B_1 ;

V (Apriv);

fin

B sera reveillé par A en exécutant V(Bpriv), A_2 et B_1 se déroulent en parallèle.

Si A_2 se termine avant B_1 , A se bloque derrière Apriv et ne sera reveillé qu'en

qu'en fin de B_1 par l'exécution de $V(A_{priv})$. Si au contraire B_1 se termine le premier, l'activation par B sera mémorisée (c.à.d. $A_{priv}=1$) et A ne se bloquera pas entre A_2 et A_3 par l'opération $P(A_{priv})$.

Ce mécanisme a marqué une étape importante dans le domaine de la synchronisation ; il traduit la coopération des processus, mais ne fournit aucun dispositif de communication; il est utile lorsqu'un processus ignore l'identité des processus avec lequel il coopère.

B.2. Sections Critiques et S.C. Conditionnelles

Les solutions données par les sémaphores sont en général assez complexes et incontrôlables à la compilation. Ce qui a conduit à forcer syntaxiquement l'écriture d'une section critique.

BRINCH et HOARE[5,17] ont introduit le concept de région critique, pour exprimer l'exclusion mutuelle dans une seule structure de la forme :

région V do S (ou with V do S) où :

V est la ressource partagée (c'est en fait une liste de variables)

S est le corps de la S.C. (S peut être une ou plusieurs instructions)

Signification :

Si un processus demande l'exécution de S en S.C. portant sur V et V étant occupée, alors il se met en attente. Sinon (c.à.d. V libre), V devient alors occupée ; il exécute S et libère la S.C.

La libération de la S.C. consiste à rendre V libre et relancer l'exécution des processus en attente de V, s'il y en a.

Cet outil permet l'écriture de la synchronisation dans les programmes et accorde au compilateur la possibilité de contrôler à son niveau l'accès, en exclusion mutuelle, aux variables partagées ; autrement dit, il permet au compilateur de

généraliser des appels aux procédures d'entrée - sortie de la région critique et de vérifier que la ressource V est bien utilisée uniquement à l'intérieur de la section critique.

Il manque dans cette structure, la description de certains événements, c.à.d. l'expression de certaines conditions qui doivent être réalisées avant ou après l'exécution d'une S.C. Cela a conduit BRINCH [6] à généraliser cet outil en introduisant la possibilité d'attente pour un processus qui demande l'accès à une ressource partagée par :

with V when B do S .

L'exécution de S en S.C. sur V dépend de l'état des variables manipulées dans cette section. Cet état est caractérisé par la valeur vraie de l'expression booléenne B ; sinon le processus demandeur est mis en attente de V , et V est rendue libre.

Notons d'une part que l'évaluation de la condition se fait en S.C. car ses opérandes ne sont modifiables qu'en cette section, et d'autre part que la file d'attente est associée à la variable partagée et non pas à la condition. Après qu'un processus quitte une telle section, toutes les conditions de synchronisation sont réévaluées, et si au moins l'une des conditions est vérifiée, un des processus en attente de cette condition est autorisé à entrer dans sa S.C.

L'inconvénient de cet outil est le manque d'efficacité du fait que d'une part, tous les processus sont suspendus dans une même file d'attente et que d'autre part, toutes les conditions sont à réévaluer lors de la relâche de la S.C.

B.3. Moniteurs

Un moniteur est un mécanisme basé sur l'idée de DIJKSTRA qui suggère de régler les problèmes de synchronisation entre n processus par le $(n + 1)^{\text{e}}$ processus qu'il appelle secrétaire. La définition du moniteur entraîne une nouvelle structuration qui est de séparer la définition des opérations de synchronisation de leurs utilisations.

Un moniteur, défini par HOARE ([18]), est une collection de données et de procédures manipulant ces données ; seules les procédures du moniteur accèdent à ces données. Ces procédures sont accessibles de l'extérieur et elles sont exécutées en exclusion mutuelle les unes par rapport aux autres. Un moniteur constitue donc une ressource critique. Sa déclaration est de la forme suivante :

```

        nom du moniteur : moniteur

début
    [
        déclarations des données locales
    ]
    procédure nom de proc (par form),
        début
            Corps de procédure
        fin;
    [
        déclarations d'autres procédures
    ]
    Initialisation des données locales

fin.
```

Un nouveau type de variables conditions a été introduit pour pouvoir suspendre et reprendre l'exécution des processus dans certains cas. Deux opérations peuvent être effectuées sur ces variables. Elle sont de la forme :

nom de condition .wait; (1)

nom de condition .signal; (2)

(1) a pour effet de suspendre le processus dans la file d'attente correspondante à cette condition et de relâcher l'exclusion sur le moniteur ; ceci permet éventuellement à un autre processus de prendre le contrôle et de venir relâcher un processus bloqué en exécutant (2).

B.3.1. Implémentation

HOARE a prouvé que le moniteur peut être implémenté par les sémaphores et réciproquement; pour cela, on utilise pour chaque moniteur un sémaphore "mutex", initialisé à 1, pour assurer l'exclusion mutuelle sur le moniteur. L'entrée au moniteur se fait par P(mutex) et la sortie par V(mutex).

Un sémaphore "urgent", initialisé à 0, ainsi qu'un compteur UC sont utilisés pour satisfaire la priorité des processus bloqués à l'intérieur du moniteur sur ceux qui veulent prendre le contre de l'extérieur .

De même qu'à chaque condition, est associé un sémaphore "condsem" (initialement nul) et un compteur CC.

* Implémentation de l'opération condition.wait :

début

CC := CC+1;

Si $UC > 0$ alors V(urgent)

sinon V(mutex);

P (cond sem);

CC := CC-1,

fin

Implémentation de l'opération cond.signal :

début

UC : = UC+1

Si CC > 0 alors

début

V(cond sem) ; P(urgent)

fin

UC : = UC-1

fin

Remarque :

Il est préférable d'après BRINCH, que l'opération signal soit la dernière instruction du corps d'une procédure; ceci pour supprimer les variables urgents et UC ainsi que les opérations définies sur ces variables.

B.3.2. Implémentation de KESSELS [21]

KESSELS a donné une autre implémentation de l'attente conditionnelle pour que celle-ci ait plus d'efficacité, en déclarant toutes les conditions à l'entrée du moniteur. Cette implémentation entraîne les avantages suivants :

a) Les conditions ne peuvent plus dépendre des données locales du processus exécutant.

b) Après avoir relâché le contrôle par un processus, seules les conditions qui ont des processus en attente seront évaluées.

Toute condition se déclare ainsi : < identificateur > : condition < expression booléenne >. Ces conditions sont manipulées par la primitive wait, suivie du nom de la condition concernée. L'exécution de wait invoquera

l'évaluation de l'expression booléenne correspondante, le processus qui l'exécute continue si cette condition est vraie, sinon il se met en attente. On utilise trois tableaux : B , Sem et w_p de N composantes. N étant le nombre de conditions, telles que, pour toute condition i :

- $B[i]$ indique l'expression booléenne correspondante à la condition i
 - $Sem[i]$ est un sémaphore associé à cette condition
 - $w[i]$ compte le nombre de processus bloqués sur cette condition.
- $Sem[0]$ est réservé pour contrôler l'accès au moniteur.

Entrée du moniteur se fait par $P(Sem[0])$;

$wait(i)$:

si $\neg B[i]$ alors

début

$w_p[i] := w_p[i] + 1;$

$get(next);$

$V(sem[next]);$

$P(sem[i]);$

$w_p(i) := w_p[i] - 1;$

fin

où get est une procédure qui détermine l'index d'une condition remplie sur laquelle un ou plusieurs processus sont bloqués; elle produit la valeur 0 autrement.

Sortie du moniteur :

$get(next);$

$V(sem[next]);$

B.4. Module de Contrôle

Parmi les critiques faites aux mécanismes de synchronisation, il est à noter qu'il n'existe pas d'analyse globale des contraintes , pour l'ensemble des processus qui partagent une variable ou une ressource commune.

Ceci constitue le souci de la plupart des concepteurs de systèmes. Aussi ROBERT et VERJUS [34,35] ont-ils été conduits à introduire la notion du module qui est basée sur deux principes :

- la séparation de la description du contrôle de celle des processus à contrôler,
- la définition d'un langage formel très proche de la langue d'expression des problèmes pour exprimer les règles de contrôle.

On distingue alors, deux types de module : un premier module appelé "module de contrôle" dans lequel on regroupe des règles de synchronisation, et un second module "module de traitement", dans lequel on trouve les procédures, ainsi que les données qu'elles partagent. On impose aux processus d'exécuter une procédure "P" entre deux points d'intervention du mécanisme de contrôle d'une façon implicite. Ces points sont l'appel et le retour de procédure. Pour satisfaire cette exigence de séparation, on définit un certain nombre de compteurs pour exprimer les conditions de synchronisation, selon la classification des problèmes :

1°) une première classe regroupe les problèmes qui requièrent seulement des informations sur l'activité d'une opération. On définit alors un compteur $\gamma(P)$ qui enregistre le nombre d'instances de la procédure P en activité. L'exclusion mutuelle est typique de cette classe de problème.

2°) Une classe incluant la première, exige la connaissance d'une

certaine "histoire" sur les opérations effectuées, et permet de savoir le nombre d'autorisations et de terminaisons d'une telle opération.

Pour cela, deux compteurs ont été définis :

$\alpha(P)$: contient le nombre total d'autorisations de l'exécution de P depuis l'initialisation

$\lambda(P)$: contient le nombre total de terminaisons de P depuis l'initialisation.

3°) Ordonnancement des procédures :

Les requêtes sont enregistrées dans des files d'attente. Pour établir des règles d'ordonnancement, telle que la priorité, un compteur $\rho(P)$ est mis en place pour calculer le nombre de requêtes enregistrées pour chaque opération.

4°) $\phi(P)$ enregistre le nombre de requêtes en attente de la procédure P.

Il est à noter qu'à tout instant on a :

$$* \quad \gamma(P) = \alpha(P) - \lambda(P)$$

$$\# \text{ activité } (P) = \# \text{ autorisation } (P) - \# \text{ terminaison } (P)$$

(le signe $\#$ signifie "nombre de")

$$* \quad \phi(P) = \rho(P) - \alpha(P)$$

$$\# \text{ requête en attente } (P) = \# \text{ requête } (P) - \# \text{ autorisation } (P)$$

Ces compteurs, initialement nuls, seront remis automatiquement à jour à chaque changement d'état du module de contrôle. Pour chaque procédure, le module de contrôle spécifiera une condition qui devra être satisfaite avant l'exécution de cette procédure.

B.5. Expressions de Chemin "path expressions"

Une nouvelle structure abstraite a été proposée par CAMPBELL et

HABERMAN [7] où la synchronisation est exprimée dans la définition du type; dans chaque définition du type, les expressions de chemin, faisant partie de sa structure, permettent d'exprimer à côté des données et procédures d'un objet, les chemins possibles que les processus doivent respecter dans leurs exécution.

Chaque type a une structure de la forme :

```

type  nom,
      [
        définition des données
      ]
      [
        path ... end
      ]
      [
        définition des opérations
      ]
end type.

```

Quatre formes syntaxiques sont utilisées pour décrire l'ordre d'exécution des procédures : Si p, q, r et s sont des procédures :

a) Exécution séquentielle

';' désigne l'opérateur de séquençement

p ; q ; r est un exemple d'une expression de chemin où les procédures p, q et r doivent être exécutées en séquence, dans l'ordre où elles se trouvent. Les procédures peuvent être invoquées par différents processus dans un ordre quelconque. Si l'invocation de q se produit la première, le processus appelant sera mis en attente jusqu'à ce que P soit exécutée.

b) Exécution sélective

',' est l'opérateur de sélection, il permet d'exprimer l'exclusion mutuelle.

L'exemple suivant : p,q,r spécifie qu'une et une seule procédure

pourrait être sélectionnée pour s'exécuter. Les processus demandant l'exécution des procédures non choisies seront bloqués jusqu'à ce qu'une nouvelle sélection soit faite.

Les deux schémas précédents peuvent être combinés en donnant une nouvelle expression.

Exemple : $P ; (q, r), s$

les parenthèses expriment les priorités.

c) Exécution répétitive

Toute procédure délimitée par "path" et "end" pourrait, une fois terminée son exécution, se répéter. L'exécution est cyclique.

Exemple : path P end indique que lorsque la procédure P a été exécutée, celle-ci pourrait recommencer tant qu'elle serait invoquée.

d) Exécution simultanée

Elle est représentée par des accolades " $\{ \}$ "

$\{ P \}$, permet à plusieurs processus d'exécuter simultanément la procédure P.

Une expression de chemin peut être représentée par un graphe traduisant un automate d'état fini où l'état initial et l'état final sont confondus.

Les arcs sont les procédures et les noeuds représentent les états.

On trouve dans [7] un algorithme d'implémentation des 'Path' en fonction des sémaphores.

Ce mécanisme de synchronisation présente le même avantage que les moniteurs à savoir de regrouper dans une même structure de données toutes les informations concernant la synchronisation. Il semble être plus structurant

car il ne possède aucun moyen explicite pour exprimer les attentes ; autrement dit, les opérations wait et signal du moniteur disparaissent.

B.6. "Serializer" [1]

est un outil de synchronisation qui encadre la ressource dans une seule structure, pour protéger son intégrité, en ordonnant les opérations effectuées par les processus. La synchronisation est exprimée en terme d'événements ordonnés de sorte que tout événement prenant le contrôle du "serializer" doit être suivi par un autre qui le relâche; ceci est utilisé pour assurer l'exclusion mutuelle dans cette structure.

Pour utiliser une ressource par un processus, le "serializer" contrôle l'occurrence de toute une séquence d'événements :

1°) "Serializer-request" : le 'serializer' reçoit un message de l'extérieur envoyé par le processus et destiné à la ressource.

2°) Événement "ENTRY" : lorsque la requête aboutit à cet événement, il est permis au processus de prendre le contrôle et d'entrer dans cette structure.

3°) Événement "GUARANTEE" : le processus sera bloqué jusqu'à ce que certaines conditions soient vérifiées, tout en relâchant l'exclusion sur le "serializer".

4°) Événement "established" : est un événement résultant du précédent avec la valeur de la condition "vrai". Il indique que le processus est autorisé à utiliser la ressource.

5°) "JOIN-CROWD" : indique que le processus est en possession de la ressource.

7°) "Exit" : le processus abandonne le contrôle en quittant le "serializer".

8°) "Serializer-Reply" : un message est transmis par cet événement comme réponse à l'événement "serializer request".

Ces événements sont répartis en deux classes distinctes :

a) Les événements autorisés à prendre le contrôle :

- ENTRY, Established, Leave CROWD et éventuellement "time-out", s'il est spécifié dans la condition d'attente qui se produit dans le cas où le processus reste bloqué plus longtemps que le temps indiqué dans cette clause.

b) Ceux qui relâchent le contrôle : Guarantee, JOIN-CROWD et EXIT.

Remarquons que dans cette structure, la condition d'attente est exprimée par une commande notée : Guarantee <condition> . Un processus qui exécute cette primitive, en trouvant la condition non vérifiée, se met en attente en relâchant l'exclusion sur le "serializer"; il ne pourrait être relâché que si les trois conditions suivantes étaient vérifiées :

1°) la valeur de la condition est vraie;

2°) aucun autre processus n'est en possession du contrôle; autrement dit un des trois événements se produit : JOIN-CROWD, EXIT ou GUARENTEE.

3°) il est le premier de la file d'attente, possédant la priorité la plus élevée s'il est dans une file d'attente avec priorité; ceci est dû à la politique choisie pour ordonner les processus bloqués.

Remarques :

- Les processus bloqués sur conditions sont plus prioritaires que ceux qui veulent accéder au Serializer de l'extérieur; ceci lui permet de répondre à toutes les requêtes reçues.

- Dans la déclaration du "serializer", il y a un nouveau type "CROWD" qui permet de compter et de savoir quels processus sont en possession de telle ressource.

- La possibilité d'inclure d'autres déclarations de "serializer" dans une telle structure; ce qui est inutile dans le cas du moniteur.

B.7. Réseaux de Petri (R. d. P.)

Dans ce paragraphe, nous donnons des définitions abstraites extrait de [28] et nous verrons sur des exemples comment représenter quelques problèmes de synchronisation à l'aide de ces réseaux.

Définitions

Un R. d. P. est un quadruplet $R = (P, T, \alpha, \beta)$ tel que :

P est l'ensemble d'objets appelés places ($P \neq \emptyset$)

T est l'ensemble d'objets appelés transitions; $T \neq \emptyset$ et $P \cap T = \emptyset$

α est une relation d'incidence avant ($\alpha \subseteq P \times T$)

β est une relation d'incidence arrière ($\beta \subseteq T \times P$)

Représentation

On représente un R. d. P. par un graphe bipartiorienté :

les noeuds du graphe sont les places représentées par des cercles, et les transitions par des barres.

Il y a un arc de la place p_s à la transition t_j $\underline{SSI} (p_s, t_j) \in \alpha$

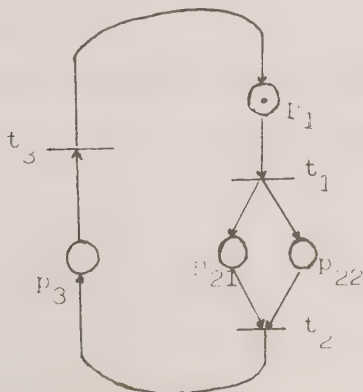
de même il y a un arc de la transition t_r à la place p_w $\underline{SSI} (t_r, p_w) \in \beta$

Notation :

$${}^{\cdot}t = \{ p \in P / (p, t) \in \alpha \}$$

$$t^{\cdot} = \{ p \in P / (t, p) \in \beta \}$$

Exemple :



$${}^{\cdot}t_1 = \{ p_1 \} = t^{\cdot}_3$$

$${}^{\cdot}t_2 = \{ p_{21}, p_{22} \}$$

$$t^{\cdot}_2 = \{ p_3 \} = {}^{\cdot}t_3$$

Marquage d'un R.d.P.

Un marquage M de R est défini comme étant une application de P dans N .

Remarque : M est un vecteur de N^n dans le cas où $|P| = n$ t.q

$$M = (m_1, m_2, \dots, m_n) \text{ avec } m_i = M(p_i)$$

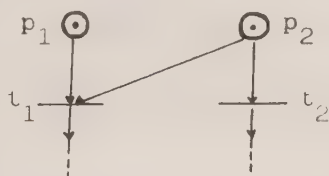
Définitions

Une transition t est dite déclenchable (tirable ou validée) par le marquage M SSI toute place d'entrée de t contient au moins une marque.

Autrement dit SSI $\forall p \in {}^*t$, on a $M(p) \geq 1$

* Des transitions déclenchables sont en conflit si la soustraction des marques en entrée de l'une d'elles rend les autres indéclenchables.

Exemple :



t_1 et t_2 sont dites en conflit du fait que le déclenchement de l'une rend l'autre indéclenchable.

Opération sur les marquages

On passe d'un marquage M_i à un marquage suivant M_j par : $\forall p \in P$

$$M_j(P) = \begin{cases} M_i(P) - 1 & \text{si } p \in {}^*t \text{ et } ({}^*t \cap t') \\ M_i(P) + 1 & \text{si } p \in t' \text{ et } ({}^*t \cap t') \\ M_i(P) & \text{sinon} \end{cases}$$

Cette opération est appelée "mise à feu", elle n'est définie que pour les marquages qui valident la transition t , et on note $MAF_t(M_i) = M_j$

Définitions

- Un réseau est dit borné si pour tout marquage (obtenu à partir du marquage initial), la marque de chaque place est bornée.

- Un réseau est dit vivant si pour tout marquage, toutes les transitions sont déclenchables (pas d'interblocage)

(exemple : réseau représenté par un paragraphe fortement connexe est un réseau vivant)

Réseaux de Petri généralisés (R.d.P.G.)

Définition

Un R.d.P.G. est un doublet $R_G = (R, \varphi)$ tel que :

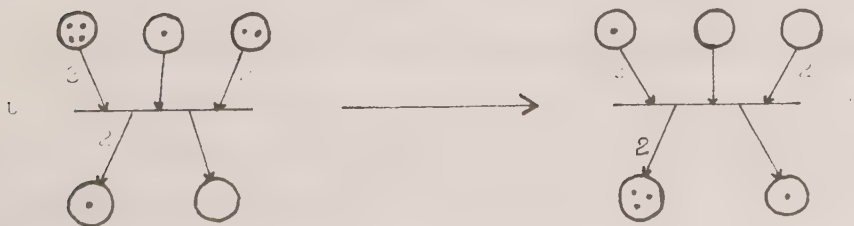
- $R = (P, T, \alpha, \beta)$ et

- $\varphi : U = \alpha \cup \beta \longrightarrow \mathbb{N}$ (U est l'ensemble des arcs de R).

Chaque arc $u \in U$ comporte comme renseignement $\varphi(u)$ appelé poids de l'arc.

- Une transition t est déclenchable SSI chacune de ses places d'entrée contient un nombre de marques au moins égal au poids de l'arc qui la connecte à t . Sa mise à feu (si elle est déclenchable) consiste à enlever de chaque place d'entrée un nombre de marques égal au poids de l'arc qui la connecte à t , et à déposer dans chaque place de sortie un nombre de marques égal au poids de l'arc qui la relie à t .

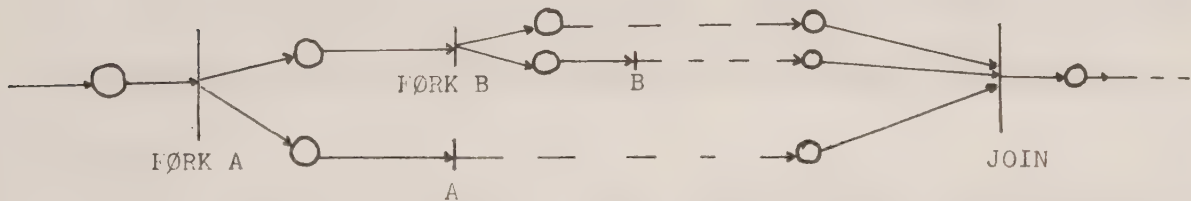
Exemple :



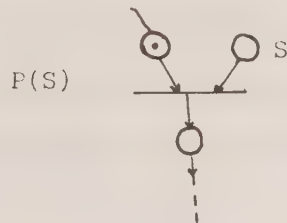
Les réseaux de Petri peuvent être utilisés pour modéliser le parallélisme et pour schématiser les solutions des problèmes de synchronisation, les ressources sont associées aux places, elles seront consommées ou produites par des actions (associées aux transitions). La disponibilité de ces ressources est liée aux marques situées dans les places correspondantes. Ces réseaux généralisent de nombreuses primitives de synchronisation.

Exemples :

1°) FØRK et JOIN [15]

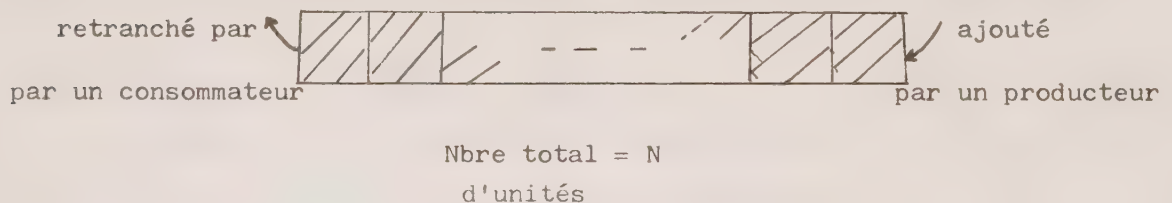


2°) Sémaphore (exclusion mutuelle)



3°) Soient une calsse de producteurs et une autre de consommateurs qui travaillent sur un tampon de N unités :

- chaque opération de production ajoute une unité en fin de zone
- chaque opération de consommation retranche une unité en tête de zone.



Si on utilise la méthode de synchronisation de ROBERT ET JURJUS, on écrit :

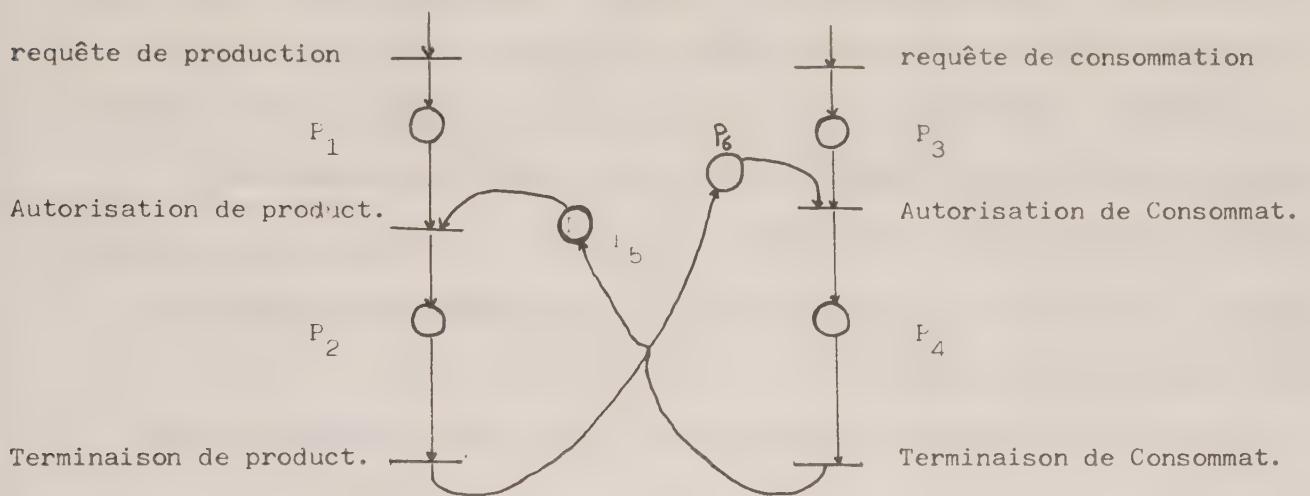
condition de production : $\# \text{ Autorisation (production)} < \# \text{ Terminaisons (Cons)} + N$

condition de consommation : $\# \text{ Autorisation (Cons)} < \# \text{ Terminaisons (Prod)}$.

Ces deux conditions généralisent l'invariant du problème :

$$(1) \# \text{ Autorisations (prod)} - \# \text{ Terminaisons (Cons)} \leq N$$

$$(2) \# \text{ Terminaisons (prod)} - \# \text{ Autorisations (Cons)} \geq 0$$



Notons que le tampon est initialement vide et donc c'est aux producteurs de commencer à le remplir et ensuite les consommateurs vident ce qui a été rempli.

Les énoncés invariants (1) et (2) précédents peuvent être traduits par les conditions suivantes qu'on vérifie à tout moment :

$$M(P_6) - M(P_4) \geq 0$$

$$M(P_5) - M(P_2) \leq N$$

On donne une définition d'extension des R.d.P. qui exprime la priorité dans certaines classes de problèmes de synchronisation.

. Réseaux à arcs Inhibiteurs (R.d.P.Z.)

Définition

Un R.d.P.Z. est un doublet $R_z = (R, \tau)$ tel que :

$R = (P, T, \alpha, \beta)$ et

$\tau \subseteq P \times T$ avec $\tau \cap \alpha = \emptyset$

τ est appelé ensembles d'arcs inhibiteurs.

- Ces arcs sont utilisables seulement en entrée d'une place vers une transition.
- La présence d'un arc inhibiteur, sortant d'une place p , rend toute transition t connectée à p indéclenchable tant que p possède des marques.

Ces arcs sont différenciés des autres arcs par un petit cercle placé aux extrémités incidentes aux transitions.

Exemple : problème de lecteurs-rédacteurs avec priorité aux rédacteurs [10]

Conditions sur chaque opération :

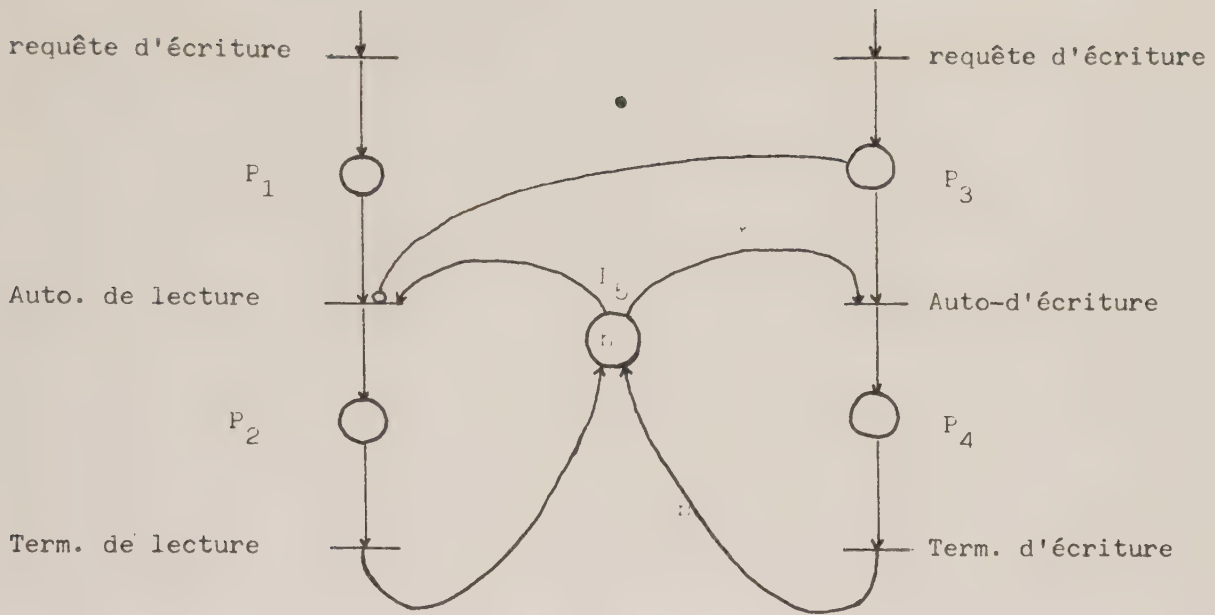
a) une lecture peut être effectuée SSI il n'y a aucun écrivain actif et aucun écrivain en attente :

Condition de lecture : $(\neq \text{Activité (écriture)} = 0) \wedge (\neq \text{requête en attente (écriture)} = 0)$

b) Une écriture peut être effectuée SSI il n'y a ni lecteurs, ni écrivains actifs.

Soit : $\left[(\neq \text{Activité (écriture)}) \wedge (\neq \text{Activité (écriture)}) \right] = 0$

la condition d'écriture.



Invariant du problème

On vérifie à tout moment :

$$M(P_2) + M(P_3) + nM(P_4) = n$$

Ceci implique que :

$$0 \leq M(P_2) \leq n \quad (1)$$

$$0 \leq M(P_4) \leq 1 \quad (2)$$

$$M(P_2) \cdot M(P_4) = 0 \quad (3)$$

qui signifient :

- 1°) n processus au plus peuvent lire simultanément.
- 2°) 1 seul processus peut écrire à la fois.
- 3°) lecture et écriture sont en exclusion mutuelle.

CHAPITRE III

PROBLEMES BINAIRES

CHAPITRE III

PROBLEMES BINAIRES

3.1. Présentation

Soit E l'ensemble des objets sur lesquels on travaille, et soit g une application de $E \times E$ dans E ; g sera appelée action; ainsi g pourra être l'addition des réels, comme elle pourra être l'opération qui, à partir de deux monotonies, forme la monotonie interclassée.

On notera par E_n l'ensemble des suites formées de n objets de E . Soit h

l'application de E_n dans E $\left\lfloor \frac{n+1}{2} \right\rfloor$ définie à partir de g , par :

. Si S est la suite $x_1, x_2, \dots, x_{2p}$

$$h(S) = g(x_1, x_2), g(x_3, x_4), \dots, g(x_{2p-1}, x_{2p})$$

. Si S est la suite $x_1, x_2, \dots, x_{2p}, x_{2p+1}$

$$h(S) = g(x_1, x_2), g(x_3, x_4), \dots, g(x_{2p-1}, x_{2p}), x_{2p+1}$$

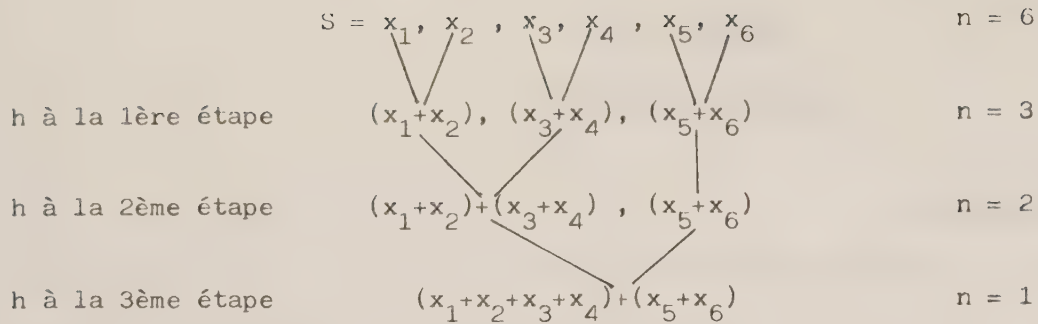
Le but est d'appliquer h successivement jusqu'à ce que le résultat soit une suite d'un seul objet et d'exécuter en parallèle, les actions de chaque étape.

3.1.1. Exemple :

On prend pour g l'addition réelle telle que :

$$\begin{array}{ccc} g : R \times R & \longrightarrow & R \\ (x, y) & & x+y \end{array}$$

Soit S la suite de six éléments $x_1, x_2, x_3, x_4, x_5, x_6$



3.2. Programmation Séquentielle

La programmation séquentielle de ce type de problème peut être exprimée selon les schémas suivants :

1°) Schéma récursif :

Soit S est la liste d'éléments de la suite, on écrit :

$h(S) =$

début

- constitution des couples
- action sur chaque couple
- la liste S devient S'
- si cardinal (S') > 1 alors h(S') fsi;

fin.

2°) Schéma itératif :

Si S est le tableau qui contient la suite à n éléments, on écrit alors :

Tant que $n > 1$ faire

début

pour $I := 1$ (pas 2) Haut $(n \text{ div } 2) \times 2$ faire

début

$S \left(\left\lfloor \frac{i+1}{2} \right\rfloor \right) := g(S(i), S(i+1))$

fin

Si n est impair mettre S(n) en S $\left(\frac{n+1}{2} \right)$;

$n := (n+1) \text{ div } 2$

fin.

Notre objectif est d'analyser, dans une optique MIMD, comment exprimer le parallélisme dans ces problèmes. Nous commençons d'abord par une analyse pour des machines SIMD.

3.3. Exemple de programmation sur ILLIAC IV

Les machines SIMD sont, en général, des machines vecteurs; les données sont structurées en vecteurs sur lesquels opèrent les processeurs; l'ILLIAC IV est un exemple de réalisation. Une telle machine utilise jusqu'à 256 processeurs, partitionnés en quatre quadrants de 64, sous le contrôle d'une unité (CU) centralisée. Chaque processeur élémentaire noté PE, qui est une unité arithmétique et logique (ALU) sophistiquée, possède six registres programmables :

A : est l'accumulateur

B : registre du deuxième opérande

R : pour transmettre une information d'un PE à un autre

S : est un registre intermédiaire

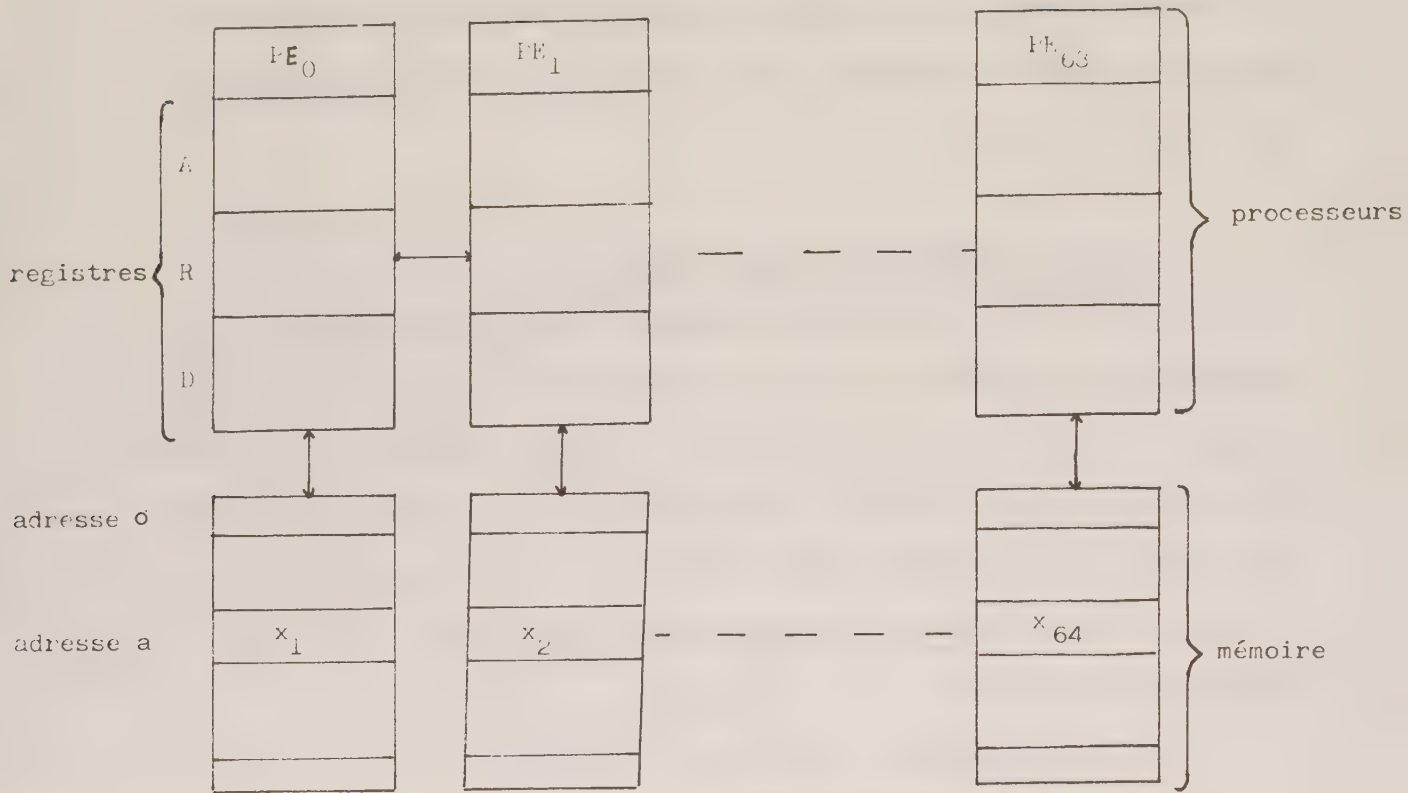
X : le registre d'index

D : contrôle l'état du processeur (actif ou non).

Notons que le traitement parallèle des problèmes binaires convient pour ce type de machine, car tous processeurs actifs peuvent faire, au même moment, la même action g. Pour voir comment exprimer les problèmes binaires sur ce type de machine, on donne un exemple de la somme de $n(=64)$ nombres sur ILLIAC IV [4].

(pour $n > 64$, le traitement est analogue, il dépendra de l'organisation des données en mémoire).

On suppose que les données sont chargées, en mémoire, à l'adresse "a" de la forme suivante :



Algorithme de programmation :

- 1°) Activer tous les PE_i (i=0,1,...,63)
- 2°) Charger l'accumulateur "A" de chaque PE_i (i=0, à 63) à partir de l'adresse a,
- 3°) $k \longleftarrow 0$
- 4°) Ranger l'accumulateur "A" dans le registre "R" pour tous processeurs PE_i (i=0 à 63)
- 5°) décaler à droite le registre R de 2^k , pour les PE_i (i=0 à 63)
- 6°) $j \longleftarrow 2^k - 1$
- 7°) désactiver les PE_i (i= 0 à j)

8°) Mettre $A \leftarrow A+R$ uniquement pour les processeurs actifs (de $j+1$ à 63)

9°) $k \leftarrow k+1$

10°) Si $k < 6$ allera 4°).

A la fin de cet algorithme, on trouve la quantité $\sum_{k=1}^{i+1} x_k$ dans l'accumulateur

de chaque PE_i et en particulier, le dernier processeur (PE₆₃) contient le résultat.

Le calcul suit le schéma énoncé en 3.1.1.

3.4. Analyse de ces problèmes pour de machines MIMD

La traduction du schéma séquentiel précédent (schéma itératif du 3.2.2.) en schéma parallèle (à l'aide des instructions existantes qui permettent d'exprimer le parallélisme) consiste à activer $\left\lfloor \frac{n}{2} \right\rfloor$ processeurs à chaque étape.

Nous faisons deux remarques à cet égard :

1°) On garde la notion d'étape; autrement dit, il faut attendre que tous les processus d'une étape aient terminé leurs exécutions pour entamer l'étape suivante.

2°) A la fin de chaque étape, tous les processus se désactivent et restituent leurs zones de travail. Ils seront ensuite recréés à l'étape suivante, et s'allouent de nouvelles zones.

Nous pouvons envisager une solution qui consiste à "casser" les étapes. Chaque fois un processeur commence un travail, il reste actif tant qu'il y a des tâches non exécutées. Cela nous amène donc à résoudre le problème lié à la gestion des processus et à leur contrôle.

3.4.1. Gestion des processus

Nous proposons un outil du type FØRK et JOIN, qui permet de contrôler les processeurs actifs. Il ne laisse "continuer" à l'étape suivante, que le nombre nécessaire et permet à ceux qui sont inutiles de se libérer, ceci dépend bien entendu, du nombre de processeurs disponibles dans la machine. Pour cela, nous définissons deux primitives notées "LFØRK" et "LJOIN" qui mettent en jeu un ensemble de paramètres (compteurs) initialisés par la première primitive; ils seront ensuite testés et modifiés par l'exécution de la deuxième. Nous supposons que chaque processeur possède un registre spécial pouvant adresser l'ensemble de ces compteurs que nous enonçons ci-après.

Pour une étape donnée :

NE : contient le nombre d'exécutions

PE : contient le nombre de processus à exécuter

PR : contient le nombre de processus qui reste à exécuter

PA : enregistre le nombre de processeurs actifs.

Remarque : $PR = PE - PA$

PS : contient le nombre de processus à exécuter à l'étape suivante

CP : compte le nombre de processeurs en vue de leur libération.

Notons que ces primitives peuvent avoir, en paramètres, un descripteur qui comporte :

n : nombre de tâches

ad : adresse de la séquence à exécuter en parallèle

adsuiv: adresse où le processeur qui exécute la dernière tâche, continue en séquence.

La rencontre de la primitive LFORK permet d'initialiser les compteurs de la façon suivante :

$$\begin{aligned} NE &:= n, \quad PE := \left\lfloor \frac{n}{2} \right\rfloor, \quad PA := 1; \\ PR &:= PE - PA; \quad Cp := 0 \end{aligned}$$

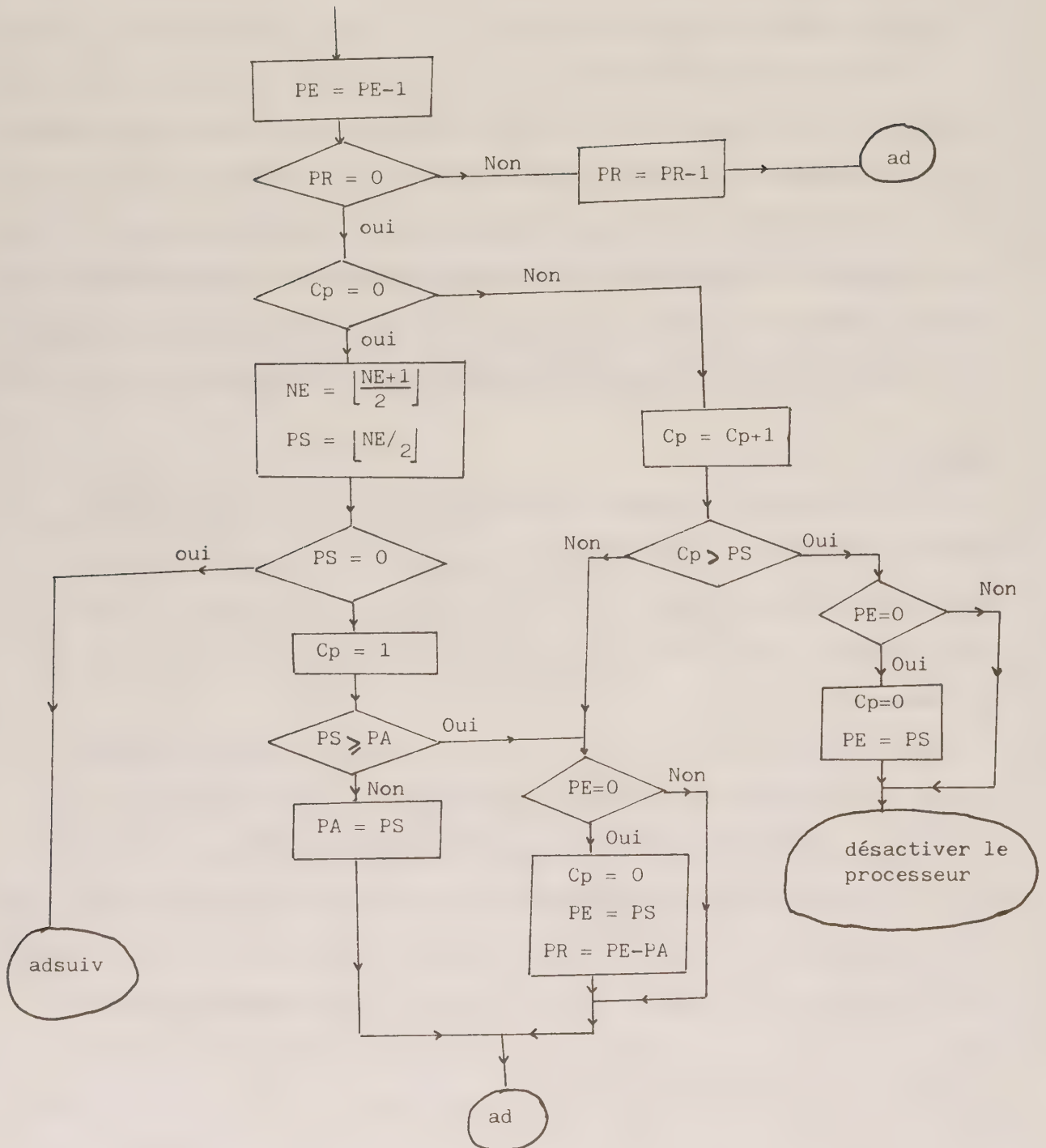
Le processeur qui exécute cette primitive, prépare l'exécution de PE processus. A chaque processeur libre est affecté un processus en incrémentant PA de 1 et en décrémentant PR de 1, tant que ce dernier soit strictement positif. Tout processeur, une fois actif, qui termine une tâche (ou processus) se contente d'exécuter la primitive LJOIN, en exclusion mutuelle avec les autres, pour déterminer s'il peut relancer l'exécution ou il se libère.

LJOIN :

L'algorithme pour contrôler les processeurs est résumé dans l'organigramme de la page 43. Etant donné, au départ, un certain nombre de processeurs actifs (en général égal à PA); si ce nombre n'est pas suffisant pour faire une étape, tout processeur, terminant une tâche, reprend l'exécution en décrémentant PR de 1.

Dans le cas contraire (c.à.d. $PR=0$), on a besoin de savoir le premier processeur qui termine une tâche pour déterminer le nombre d'exécutions à l'étape suivante. Ce processeur est spécifié par le test ($Cp=0$) ; de même, on a besoin du dernier processeur pour mettre à jour les compteurs, celui-ci est repéré par ($PE=0$). Les processeurs qui deviennent inutiles sont déterminés par le test ($Cp > PS$), ils seront désactivés par la suite.

Organigramme :



Ceci nous permet donc de donner une définition formelle d'une instruction du type "Pour tout n faire en parallèle" pour exprimer ce type de parallélisme, elle pourra être implémentée par les deux primitives LFORK et LJOIN.

3.4.2. Gestion des tâches

Après la mise en oeuvre des processus, un autre problème (lié à leur contrôle) intervient. Ceci est dû au fait que tout processus utilise deux tâches pour n'en faire qu'une seule. Ces tâches peuvent être non encore prêtes, il faudrait alors suspendre l'exécution des processus. Pour cela, on suppose qu'on a un système possédant un outil de synchronisation qui gère les attentes sur condition(s) (outil du type moniteur).

Supposons que les tâches soient rangées dans une file, cette file sera partagée par plusieurs processus; sa gestion sera simulée au problème "producteurs-consommateurs" où chaque processus prend deux états successifs: consommateur-producteur.

Au départ; tous les processus sont consommateurs et la file contient n tâches. L'exécution de chaque processus est de la forme :

consommer deux tâches action produire une seule.

L'écriture, en terme de ROBERT et VERJUS [34], des conditions d'invariance :

condition pour autoriser une consommation :

$$(1) \quad n - 2 \neq \text{Aut} (C) + \neq \text{Term} (P) > 1 \quad \text{où}$$

$\neq \text{Aut} (C)$ signifie le nombre d'autorisation d'une consommation

$\neq \text{Term} (P)$ signifie le nombre de terminaison d'une production.

On ne fait aucune condition sur les processus "producteurs", car les "consommateurs" sont eux-mêmes qui produisent, il suffit donc que les productions

se fassent en exclusion mutuelle entre les processus.

3.5. Exemple d'application : tri par interclassement naturel

E_n = l'ensemble de n monotonies, action = fusion.

Soit $A(0:m-1)$ un tableau de m éléments à trier dans un certain ordre. Ce tableau est constitué de n monotonies ($n \leq m$); une monotonie étant une suite d'éléments ordonnés de A.

Il s'agit d'interclasser ces monotonies deux par deux, de façon à obtenir le tableau 'A' trié. Une monotonie de A est déterminée par deux indices : indice de début et indice de fin; on définit (pour repérer ces indices) deux tableaux $DEB(0:n-1)$ et $FINN(0:n-1)$ de la façon suivante :

$Deb(i)$ indique l'indice du début de la monotonie $n=i$ dans le tableau A.

$Finn(i)$ indique l'indice de fin de cette monotonie.

Ces tableaux seront partagés par les processus; nous donnons une solution en terme de moniteur, en ne synchronisant que les procédures "autoriser" et "terminer" une action. Les processeurs cherchent leurs tâches, lorsqu'ils sont autorisés à consommer, en exclusion mutuelle à partir du moniteur; de même, quand ils terminent leurs exécutions, ils le consultent pour déterminer l'indice de production du résultat de leurs interclassements.

Notons que chaque processeur recopie ses deux monotonies dans une zone propre à lui (ou mémoire locale) pour assurer le rangement des résultats des autres processeurs dans le tableau commun A. Ce tableau sera géré d'une façon circulaire. Pour ranger une monotonie dans ce tableau, on doit connaître l'indice JA (indice, dans A, auquel elle sera rangée) ainsi que sa longueur L déterminée par :

$$L = \left| FINN(j) - DEB(j) \right| + \left| FINN(j) - DEB(i) \right| + 2$$

Si la monotonie en question est résultat d'interclassement de la monotonie i avec la monotonie j.

Soit contrôleur le nom du moniteur. Les processus seront contrôlés par le programme suivant.

Contrôleur : moniteur

début

Var IP, IC, IR, M, N, PLEIN : entier ;

Du produit : condition PLEIN > 1 ;

Procédure Autoriser production (Var ID, LM, IA : entier) ;

début

ID:= IP ;

IP:= IP $\oplus_N$ 1 ; (* $\oplus_N$ est une addition module N *)

IA:= IR module M ;

IR:= IR + LM ;

fin

Procédure Terminer production ;

début

PLEIN:= PLEIN + 1 ;

fin ;

Procédure Autoriser consommation (Var IM1, IM2 : entier) ;

début

WAIT du produit ;

PLEIN:= PLEIN - 2 ;

IM1:= IC ;

IC:= IC + 1 ;

Si IC = N alors (N:= N div 2 ; IC:= 0) fsi ;

IM2:= IC ;

IC:= IC + 1 ;

Si IC = N alors (N:= N div 2 ; IC:= 0) fsi

fin

(* Initialisation des variables *)

PLEIN:= N ; IP:= 0 ; IC:= 0, IR:=0

Fin du moniteur.

Variables utilisées dans le moniteur

IP : correspond à la monotonie produite (résultat d'interclassement de deux monotonies) telles que $DEB [IP]$ et $FINN(IP)$ indiquent respectivement son début et sa fin.

IC : correspond à la monotonie à interclasser.

PLEIN : est la condition d'attente pour les processus "consommateurs". Elle exprime la condition d'invariance (1) du 3.4.2.

N : est le nombre de monotonies pour une étape; le "consommateur" qui utilise la monotonie $n^{\circ} N$, change la valeur de N à $\lfloor N/2 \rfloor$.

M : le nombre d'éléments du tableau à trier.

IR : Indice de rangement du résultat de chaque processus dans le tableau A.

IM : est la longueur de la monotonie résultat d'interclassement.

Nous donnons, ci-après, un schéma de programme pour faire le tri par interclassement que nous détaillerons par la suite.

Schéma de programme prallèle :

pour tout N faire en parallèle

début du bloc

étape 1 : recherche des indices de début et fin de deux monotonies à interclasser, à partir du moniteur.

étape 2 : recopier ces monotonies dans une zone "propre" (ou mémoire locale).

étape 3 : Interclasser les deux monotonies.

étape 4 : chercher, à partir du moniteur, l'indice pour enregistrer le début et la fin de la monotonie résultat, ainsi que l'indice auquel elle doit être rangée dans le tableau A.

étape 5 : ranger cette monotonie dans A.

étape 6 : avertir le moniteur qu'il y a une monotonie prête.

fin du bloc

Nous traduisons les différentes étapes de ce schéma dans le programme suivant.

Toutefois, nous faisons quelques remarques :

1°) les tableaux A, DEB, FINN sont communs à tous les processus, les variables locales d'une procédure sont propres à chaque processus.

2°) le tableau sera trié, de façon circulaire à la fin du programme, à partir de l'indice JA du dernier processus.

début du programme

type T = Tableau [0 .. M-1] de entier, TT = Tableau [0 .. N-1] de entier ;

Var A, B, C : T ; DEB, FINN : TT ;

[déclaration d'autres variables

Procédure recopie (var A, B : T ; X, D, F : entier) ;

var I : entier ;

début

pour I:=D Haut F faire

début

B [X] := A [I] ; X:= X+1

fin ;

fin;

Procédure fusion (var B, C : T ; Deb1, Fin1, Deb2, Fin2 : entier) ;

var D, F, RG, U, V, X : entier ; S : booleen ;

début

RG:=-1 ; D:= Deb1 ; F:= Fin1 ; S: Vrai ; V:= B [deb2] ;

Tant que D ≤ F faire

début

U:= B [D] ; RG:= RG+1 ;

Si U > V Alors

début

Si S Alors

début

deb1:= D ; D:= Deb2;

F:= Fin2 ; S:= Faux

fin

Sinon

début

deb2:= D ; D:= Deb1 ;

F:= Fin1 ; S:= Vrai

fin

X:= U ; U:= V ; V:= X

fin ;

C [RG] := U ; D:= D+1

fin;

(* suite de la procédure fusion *)

Si $D > \text{Fin2}$ alors début $D := \text{Deb1}$; $F := \text{Fin1}$ fin

Sinon début $D := \text{deb2}$; $F := \text{Fin2}$ fin

Tant que $D \leq F$ faire

début

$\text{RG} := \text{RG} + 1$; $\text{C}[\text{RG}] := \text{B}[D]$; $D := D + 1$

fin

Fin de la procédure fusion

Procédure Ranger (Var C, A: T ; IA, FA : entier) ;

Var I, k : entier

début

$k := 0$; pour $l := \text{IA}$ Haut FA Faire

début

$\text{A}[I] := \text{C}[k]$; $k := k + 1$

fin

fin ;

(* programme principal *)

pour tout N faire en parallèle

Début du bloc

Contrôleur. Autoriser consommation (I,J) ;
(* étape 1 *)

$D_1 := \text{DEB}[I]$; $F_1 := \text{FINN}[I]$; $l_1 := |F_1 - D_1| + 1$ (* | signifie
 $D_2 := \text{DEB}[J]$; $F_2 := \text{FINN}[J]$; $l_2 := |F_2 - D_1| + 1$; valeur absolue *)

Si $F_1 \geq D_1$ alors recopie (A,B,O, D_1 , F_1)

Sinon début

recopie (A,B,O, D_1 ,M-1);

recopie (A,B,M- D_1 ,O, F_1)

fin;

Si $F_2 \geq D_2$ alors recopie (A,B, l_1 , D_2 , F_2)

Sinon début

recopie (A,B, l_1 , D_2 ,M-1) ,

recopie (A,B, $l_1 + M - D_2$,O, F_2) ;

fin;

(*étape 2*)

Fusion $(B, C, D_1, F_1, D_2, F_2)$; $L := l_1 + l_2$;
 (*étape 3*)

Contrôleur. Autoriser production (JD, L, JA) (étape 4)

DEB $[JD] := JA$, $JF := (JA-1) \oplus_M L$, ($\oplus_M$ = addition module M)

FINN $[JD] := JF$;

Si $JF \geq JA$ alors Ranger (C, A, JA, JF)

Sinon début ranger $(C, A, JA, M-1)$, ranger $(C, A, 0, JF)$ fin ; (étape 5)

Contrôleur. Terminer production (*étape 6*)

Fin du programme

CHAPITRE IV

TRIS SYNCHRONES

CHAPITRE IV

TRIS SYNCHRONES4.1. Introduction

Les problèmes de comparaison tels que la recherche de l'élément maximum d'un tableau, l'interclassement de deux monotonies et le tri d'un fichier consistent à appliquer une série de comparaisons sur chaque couple d'éléments. Traiter ces problèmes, sur des machines SIMD, revient à faire à chaque étape les comparaisons "disjointes" au même moment (les comparaisons disjointes sont des comparaisons appliquées aux couples qui n'ont pas d'élément commun). La complexité d'un tel algorithme est une fonction du nombre de processeurs et de la taille du fichier ; elle est exprimée en nombre de niveaux d'exécution. Un niveau est défini comme une unité de temps où les comparaisons disjointes peuvent être exécutées simultanément.

A partir des algorithmes donnés par KNUTH[22], nous présentons l'aspect récursif et donnons la programmation parallèle. Un exemple de tri, sur machine multiprocesseur du type ILLIAC IV[23], sera présenté à la fin de ce chapitre.

4.2. Tri à distance

Un algorithme a été défini par BATCHER (KNUTH [22]) qui consiste à trier, à chaque étape, les éléments d'un fichier à distance d .

Une suite $(x_i)_{i=1,n}$ est dite triée à distance d si, pour tout j ($1 \leq j \leq n-d$), on a $x_j \leq x_{j+d}$.

Les étapes de l'algorithme sont concrétisées par les valeurs d'une variable m . Cette variable prend successivement les valeurs suivantes : $2^{t-1}, 2^{t-2}, \dots, 1$ avec $t = \lceil \log_2 n \rceil$. Pour chaque valeur de m , un premier passage consiste à faire des comparaisons-échanges, par tranche de m , sur les éléments à distance m à partir de l'indice 1 ; Si $m < 2^{t-1}$, on compare ensuite les éléments à distance $2^{t-1} - m$, puis $2^{t-2} - m, \dots, m$, à partir de l'indice $m+1$.

Algorithme :

M1 : Initialisation de m

prendre $t := \lceil \log_2 n \rceil$ de façon que $2^t \geq n$.
 $m := 2^{t-1}$

M2 : tant que $m > 0$ faire

début

$q := 2^{t-1}$, $r := 0$, $d := m$,

répéter M3 jusque $(m > q)$

M3: début

$i := 0$

tant que $(i < n - d)$ faire

(si $(i \wedge m = r)$ alors

(si $X(i+1) > X(i+d+1)$ alors

échanger $(X(i+1), X(i+d+1))$

) fsi ;

) fsi ;

$i := i+1$

(* ' $\wedge$ ' est le 'and' logique portant sur la représentation binaire de i et m *)

fait,

si $q \geq m$ alors ($d := q - m$; $q := q/2$; $r := m$) fsi ;

fin ;

$m := m/2$

fin.

Les comparaisons-échanges du bloc M3 sont disjointes, elles peuvent donc se faire en parallèle. Pour une étape donnée (c.à.d. $m=2^{t-i}$), M3 est exécuté i fois ($i = 1, \dots, t$) ; le coût de l'algorithme est de $\frac{t(t+1)}{2}$ opérations .i.e.

$$\frac{\lceil \log_2 n \rceil (\lceil \log_2 n \rceil + 1)}{2} .$$

Le tri par cette méthode, est donc en $O \left((\lceil \log_2 n \rceil)^2 \right)$.

Remarquons tout particulièrement; que lorsque $m = 1$, l'algorithme interclasse deux monotonies disposées de façon qu'un élément de l'une d'elles doit être suivi d'un élément de l'autre.

4.3. Tri par Interclassement

Les algorithmes les plus efficaces sont ceux construits sur des réseaux de tri "network sorting". Un réseau consiste en un nombre d'éléments de comparaison ; chaque élément reçoit deux entrées A et B pour les trier en sorties H et L selon le schéma suivant :



Figure 1 : élément de comparaison

Parmi les algorithmes développés, nous retenons ceux de BATCHER [2] "odd-even merging" et "bitonic sorting" qui trient un fichier de n éléments en $O (\log_2^2 n)$ avec $O (n (\log_2 n)^2)$ éléments de comparaison.

4.3.1. Interclassement "pair-impair"

Cette méthode consiste à fusionner deux séquences ordonnées par interclassement des indices impairs de chaque séquence indépendamment de celui des indices pairs. On applique ensuite l'opérateur comparaison-échange au résultat d'interclassement. Ceci est réalisé selon le schéma de la figure 2.

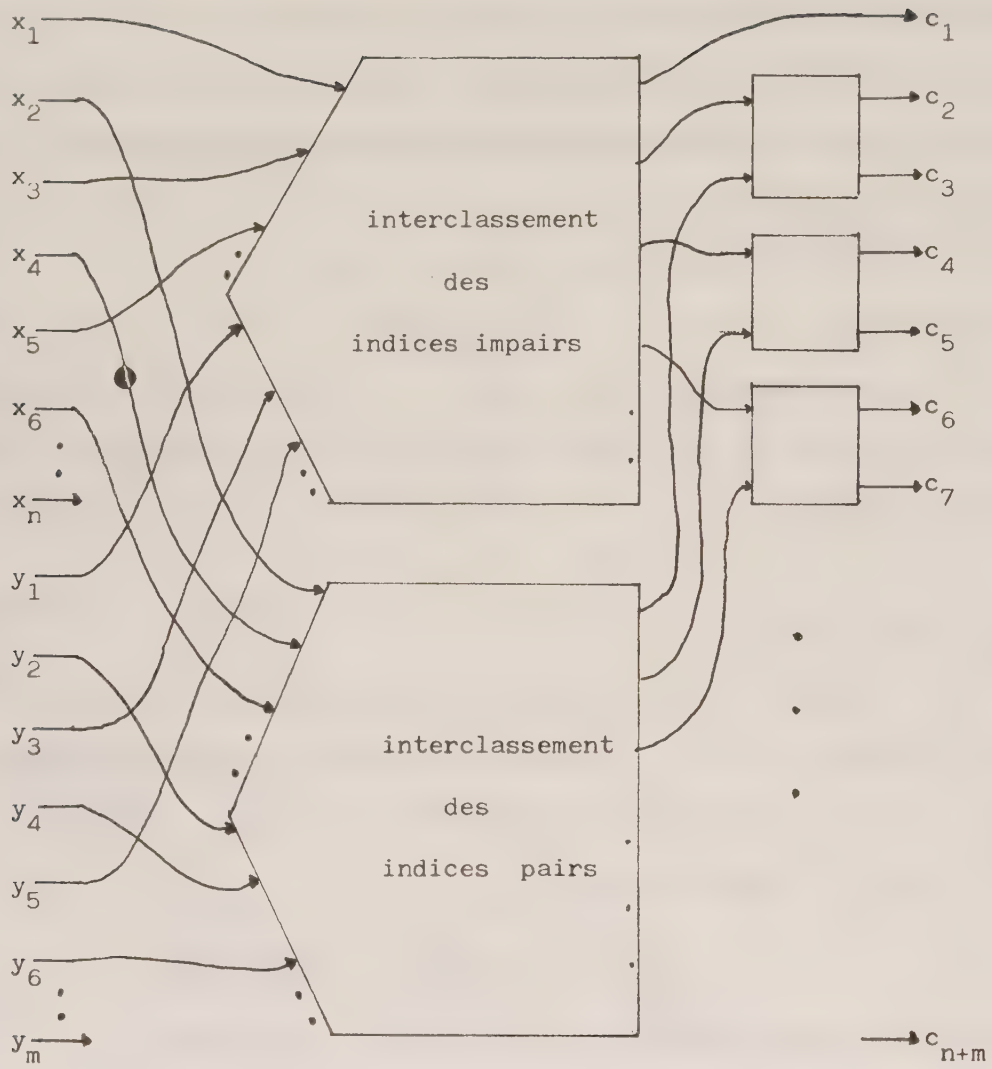


Figure 2

Schéma d'interclassement des monotonicités $(x_i)_{i=1,n}$ et $(y_j)_{j=1,m}$
 $(c_k)_{k=1,n+m}$ est le résultat trié.

Le procédé pour construire l'interclassement est récursif, nous le schématisons par la procédure suivante :

Supposons que les deux monotonies sont rangées dans une liste X l'une à côté de l'autre, la longueur de la première est n et celle de la deuxième est m .

Soit $I(j, n, k, m, F, d)$ le nom de la procédure qui interclasse la monotonie

(1) $x_j, x_{j+d}, x_{j+2d}, \dots, x_{j+(n-1)d}$ avec la monotonie

(2) $x_k, x_{k+d}, x_{k+2d}, \dots, x_{k+(m-1)d}$

j est l'indice du premier élément de la première monotonie

k est l'indice du premier élément de la deuxième monotonie

$F = n+m$

Par un appel $I(1, n, n+1, m, n+m, 1)$, la liste sera triée.

$I(j, n, k, m, F, d) =$

début

si $(n=m=1)$ alors $(CE(x_j, x_k), \text{return}))$ fsi ;

si $m=1$ alors

(Tant que $j < k$ faire

début

$CE(x_j, x_k) ; j := j+d$

fin

return)fsi ;

si $n=1$ alors

(Tant que $k \leq F$ faire

début

$CE(x_j, x_k) ; j := k ; k := k+d$

fin

return)fsi ;

$SIIG(j, \lceil \frac{n}{2} \rceil, k, \lceil \frac{m}{2} \rceil, F, 2d) ;$

$SIPD(j+d, \lfloor \frac{n}{2} \rfloor, k+d, \lfloor \frac{m}{2} \rfloor, F, 2d)$

(3) $I(j, \lceil \frac{n}{2} \rceil, k, \lceil \frac{m}{2} \rceil, F, 2d) ;$

(4) $I(j+d, \lfloor \frac{n}{2} \rfloor, k+d, \lfloor \frac{m}{2} \rfloor, F, 2d) ;$

$SCE(j, n, k, m, F, d) ;$

Fin.

où :

* CE est l'opérateur comparaison-échange

* SIIG ($j, \lfloor \frac{n}{2} \rfloor, k, \lfloor \frac{m}{2} \rfloor, F, 2d$) est une séparation d'indices impairs des suites (1) et (2) dans la moitié gauche de la liste. Elle est de la forme :

$$\underbrace{x_j, x_{j+2d}, \dots, x_{j+2(\lfloor \frac{n}{2} \rfloor - 1)d}}_{\text{éléments d'indices impairs de (1)}} \quad \underbrace{x_k, x_{k+2d}, \dots, x_{k+2(\lfloor \frac{m}{2} \rfloor - 1)d}}_{\text{éléments d'indices impairs de (2)}}$$

* SIPD ($j+d, \lfloor \frac{n}{2} \rfloor, k+d, \lfloor \frac{m}{2} \rfloor, F, 2d$) est une séparation d'indices pairs de (1) et (2) dans la moitié droite de la liste :

$$\underbrace{x_{j+d}, x_{j+3d}, \dots, x_{j+(2\lfloor \frac{n}{2} \rfloor - 1)d}}_{\text{éléments d'indices pairs de (1)}} \quad \underbrace{x_{k+d}, x_{k+3d}, \dots, x_{k+(2\lfloor \frac{m}{2} \rfloor - 1)d}}_{\text{éléments d'indices pairs de (2)}}$$

* SCE (j, n, k, m, F, d) est une série de comparaison-échange (CE) appliquée aux éléments de la première et de la deuxième séparation de la façon suivante :

on compare un élément de la première séparation avec un autre élément de la deuxième, mais les éléments de la première séparation sont comparés à partir du deuxième indice. Autrement dit, on fait :

CE (x_{j+2d}, x_{j+d}) , CE (x_{j+4d}, x_{j+3d}) , ... jusqu'à épuisement de l'une des séparations.

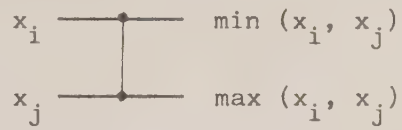
Remarque :

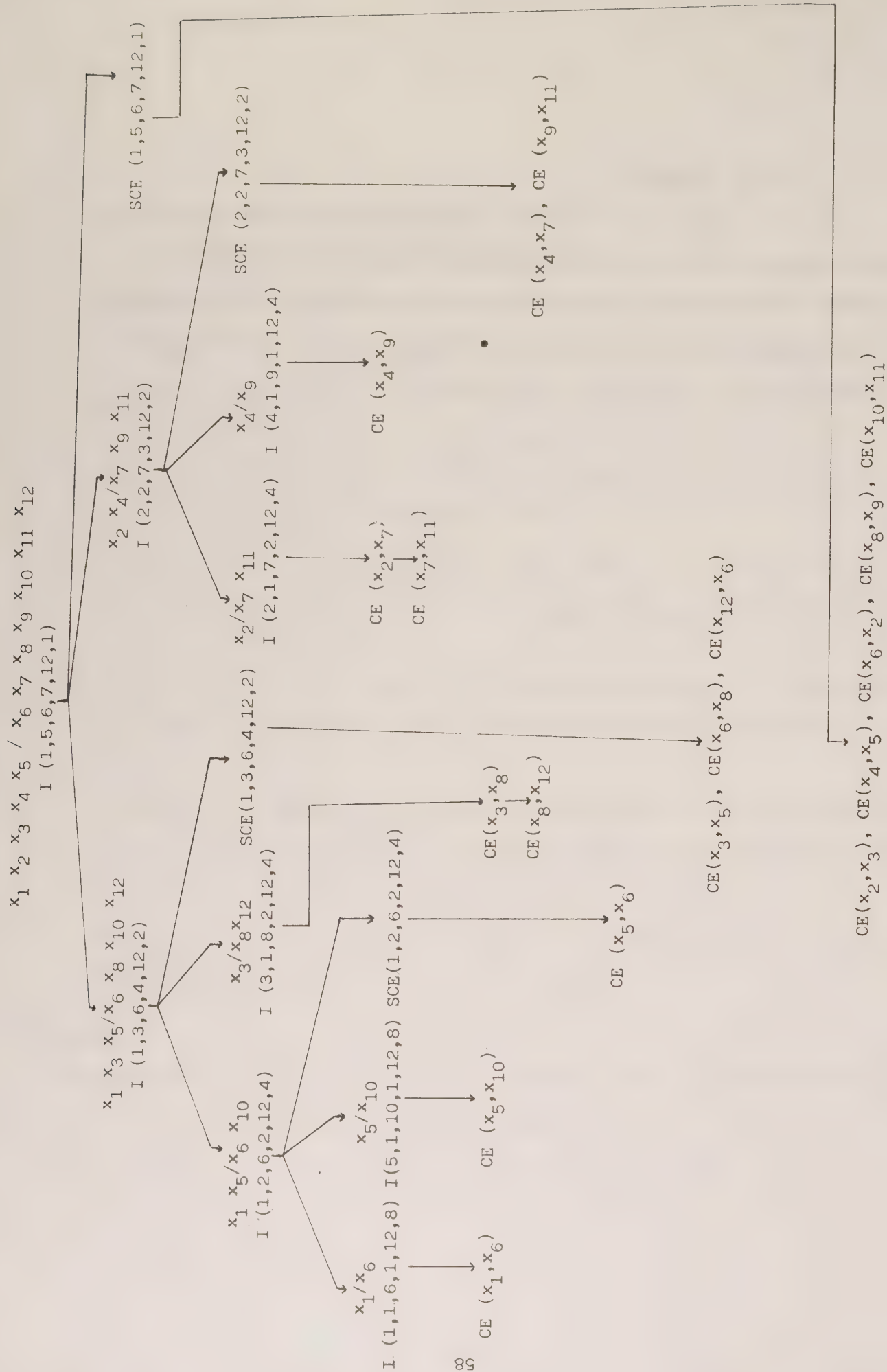
* les séparations de chaque étape se font en parallèle car les indices sont disjoints; il en est de même pour les comparaisons-échanges de SCE.

* les deux appels (3) et (4) se font en même temps à chaque étape.

4.3.1.1. exemple :

Soit à interclasser une monotonie de longueur $n=5$ avec une autre de longueur $m=7$ ($F=12$). Les différentes étapes de la procédure sont détaillées dans le schéma 3. Le schéma 4 indique le réseau de tri correspondant. Les éléments de comparaison sont représentés par des lignes verticales de cette forme :





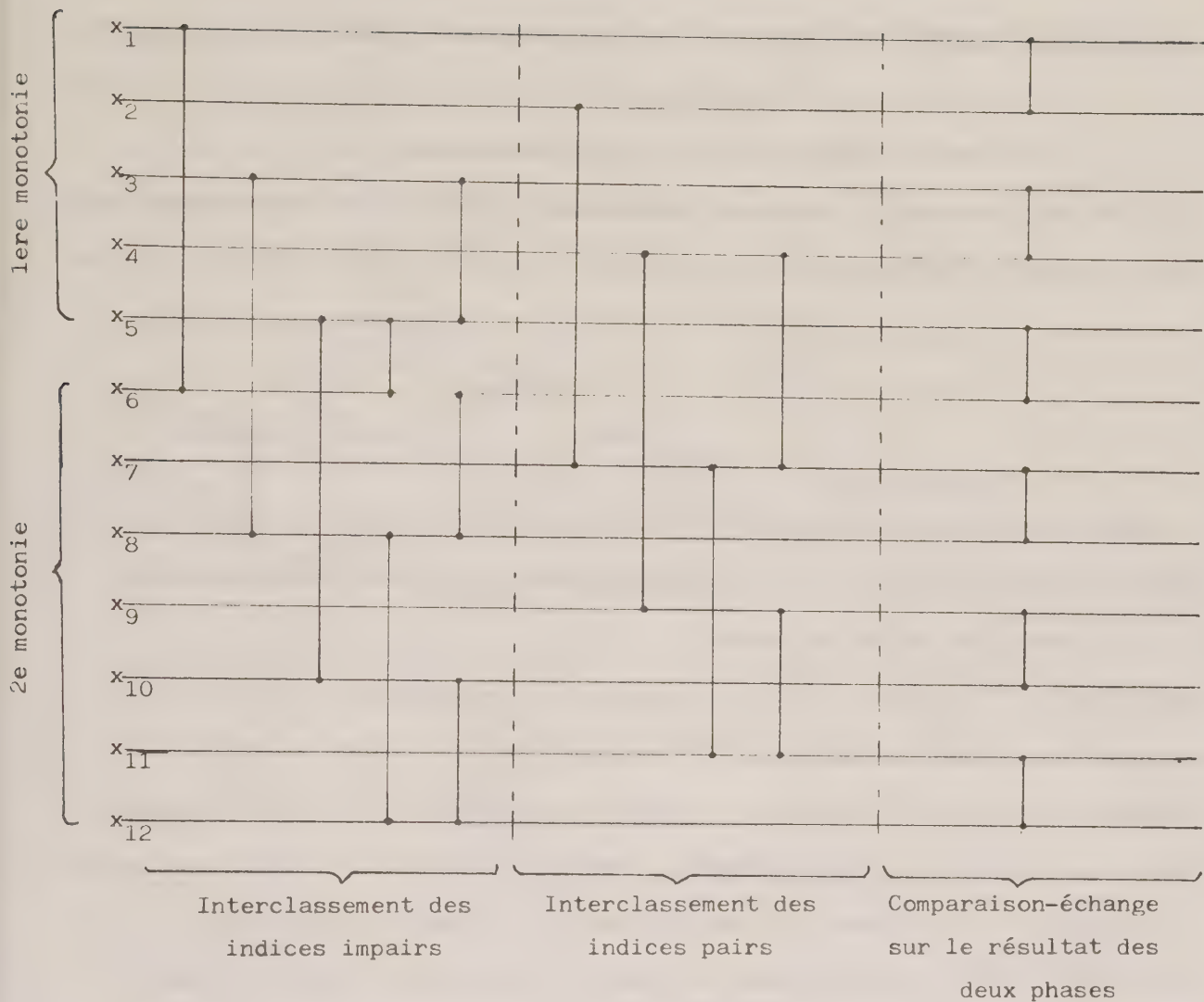


Schéma 4

4.3.1.2. Coût de la méthode

Soit $T_B(n, m)$ le nombre d'unités de temps nécessaire pour interclasser deux monotonies de longueur n et m , on écrit alors :

$$T_B(1, 1) = 1 \text{ et}$$

$$T_B(n, m) = 1 + \max \left(T_B \left(\lceil n/2 \rceil, \lceil m/2 \rceil \right), T_B \left(\lfloor n/2 \rfloor, \lfloor m/2 \rfloor \right) \right) \text{ pour } nm \geq 2$$

qui est équivalent à

$$T_B(n, m) = 1 + T_B \left(\lceil n/2 \rceil, \lceil m/2 \rceil \right) \quad \text{pour } nm \geq 2$$

Ceci implique (KNUTH [22]) que

$$T_B(n, m) = 1 + \lceil \log_2 \max(n, m) \rceil \quad \text{pour } n, m \geq 1$$

L'exemple précédent (schéma 4) utilise 4 unités de temps.

Soit $C(n, m)$ le nombre d'éléments de comparaison utilisé pour interclasser les deux monotonies ; ce nombre est solution de l'équation :

$$C(n, m) = \begin{cases} n, m & \text{si } n, m = 1 \\ C(\lceil n/2 \rceil, \lceil m/2 \rceil) + C(\lfloor n/2 \rfloor, \lfloor m/2 \rfloor) + \lfloor (n+m-1)/2 \rfloor & \text{si } n, m > 1 \end{cases}$$

On remarque que dans le cas $m=n=2^t$, l'interclassement de deux monotonies de 2^t éléments chacune nécessite $t \cdot 2^t + 1$ comparaisons qui peuvent être exécutées en $t+1$ niveaux (ou unités de temps).

Pour trier un fichier de n articles, on applique successivement cette méthode en constituant des monotonies de 2 éléments, puis de 4, ainsi de suite.

En particulier si $n=2^t$ ($t=\log_2 n$) ; on constitue d'abord 2^{t-1} monotonies de 2 éléments, ensuite 2^{t-2} de 2^2 éléments jusqu'à ce qu'on ait une seule monotonie.

A une étape k , on forme 2^{t-k} monotonies de 2^k éléments, il y a donc

$((k-1)2^{k-1} + 1) \cdot 2^{t-k}$ comparaisons qui peuvent se faire en k unités de temps ;

comme il y a t ($=\log_2 n$) étapes, le tri de $n=2^t$ éléments par cette méthode

utilise $(t^2 - t + 4) \cdot 2^{t-2} - 1$ comparaisons réparties en $\frac{t(t+1)}{2}$ niveaux

(unités de temps).

4.3.2. Interclassement "bitonic"

Définition :

Une suite $(x_i)_{i=1, q}$ est dite "bitonic" d'ordre q s'il existe un entier k ($1 \leq k \leq q$) tel qu'on ait : $x_1 \geq \dots \geq x_k \leq \dots \leq x_q$

Autrement dit c'est une juxtaposition d'une première suite décroissante suivie d'une deuxième croissante.

L'interclassement est réalisé selon le réseau de la figure 5.

Tout réseau peut être construit d'une façon récursive. Pour cela, nous définissons une procédure notée $I(j, q, d)$ qui trie une liste "bitonic" X

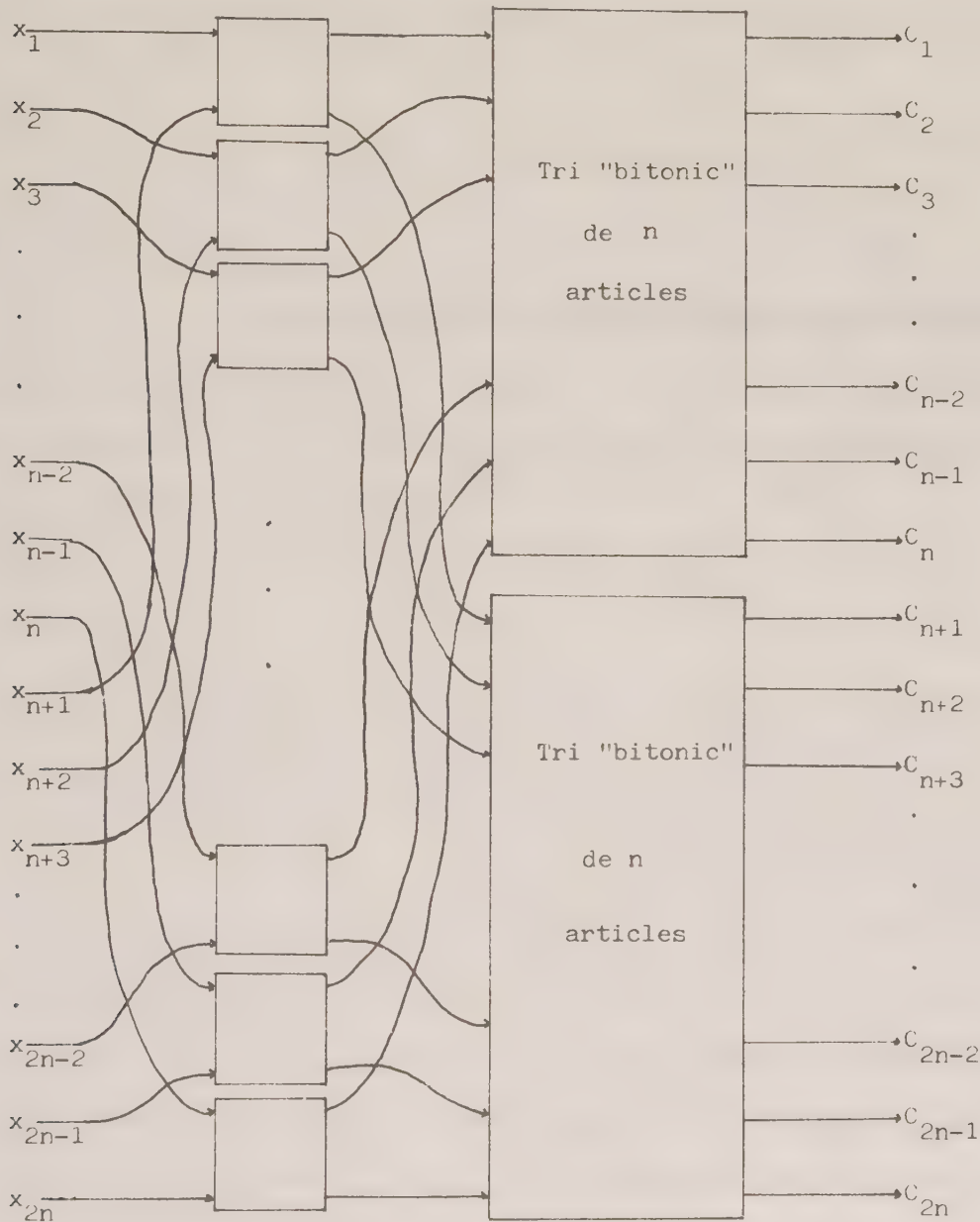


Figure 5

Réseau de tri "bitonic" d'ordre $2n$

d'ordre q à distance d , à partir de l'indice j . Les éléments de cette liste

sont : $x_j, x_{j+d}, x_{j+2d}, \dots, x_q$ (5)

Nous écrivons alors :

$I(j, q, d) =$

début

si $q = 1$ alors return fsi ;

si $q = 2$ alors (CE (x_j, x_{j+d})) ; return) fsi;

(6) $I(j, \lceil q/2 \rceil, 2d)$

(7) $I(j+d, \lfloor q/2 \rfloor, 2d)$

SCE (j, q, d)

fin

SCE est la procédure qui compare les éléments à distance d :

$SCE(j, q, d) =$

début

Pour $i := 1$ à $\lfloor q/2 \rfloor$ faire

CE ($x_{j+2(i-1)d}, x_{j+(2i-1)d}$)

fait

Remarques :

* $I(j, \lceil q/2 \rceil, 2d)$ prépare les éléments d'indices impairs de la liste bitonic (5) pour être comparés par la procédure SCE. Ces éléments sont à distance $2d$ à partir de x_j , c.à.d. :

$x_j, x_{j+2d}, x_{j+4d}, \dots, x_{2(\lceil q/2 \rceil - 1)d}$

Il en est de même pour $I(j+d, \lfloor q/2 \rfloor, 2d)$ qui prépare les éléments d'indices pairs de (5) à partir de x_{j+d} :

$x_{j+d}, x_{j+3d}, \dots, x_{(2\lfloor q/2 \rfloor - 1)d}$

Ces sous-listes sont aussi "bitonic".

* A chaque exécution de la procédure récursive I, les appels (6) et (7) seront lancés en même temps.

* Les comparaisons-échanges de la procédure SCE sont disjointes. Elles peuvent être exécutées en même temps en une seule unité de temps.

* L'interclassement "pair-impair" est un cas particulier de celui-ci ; en effet, il suffit de ranger les deux monotonies $(x_i)_{i=1,n}$ et $(y_j)_{j=1,m}$ dans la liste X de la façon suivante :

$X = (x_n, x_{n-1}, \dots, x_1, y_1, y_2, \dots, y_m)$ qui est "bitonic" d'ordre $n + m$

4.3.2.1. Exemple :

On peut prendre deux monotonies de longueur quelconque pourvu que la somme des deux longueurs soit égale à q . pour $q = 12$, l'exécution de la procédure récursive I (1,12,1) est schématisée par la figure 6. Le réseau correspondant est exprimé par la figure 7.

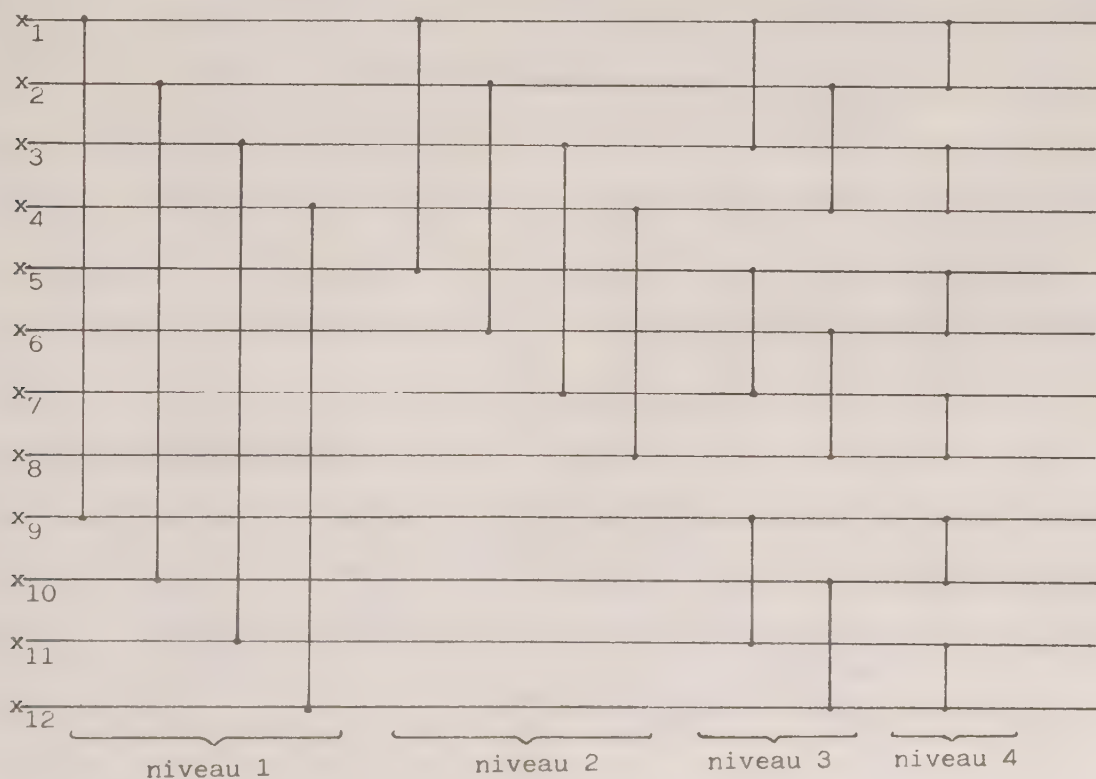


Figure 7

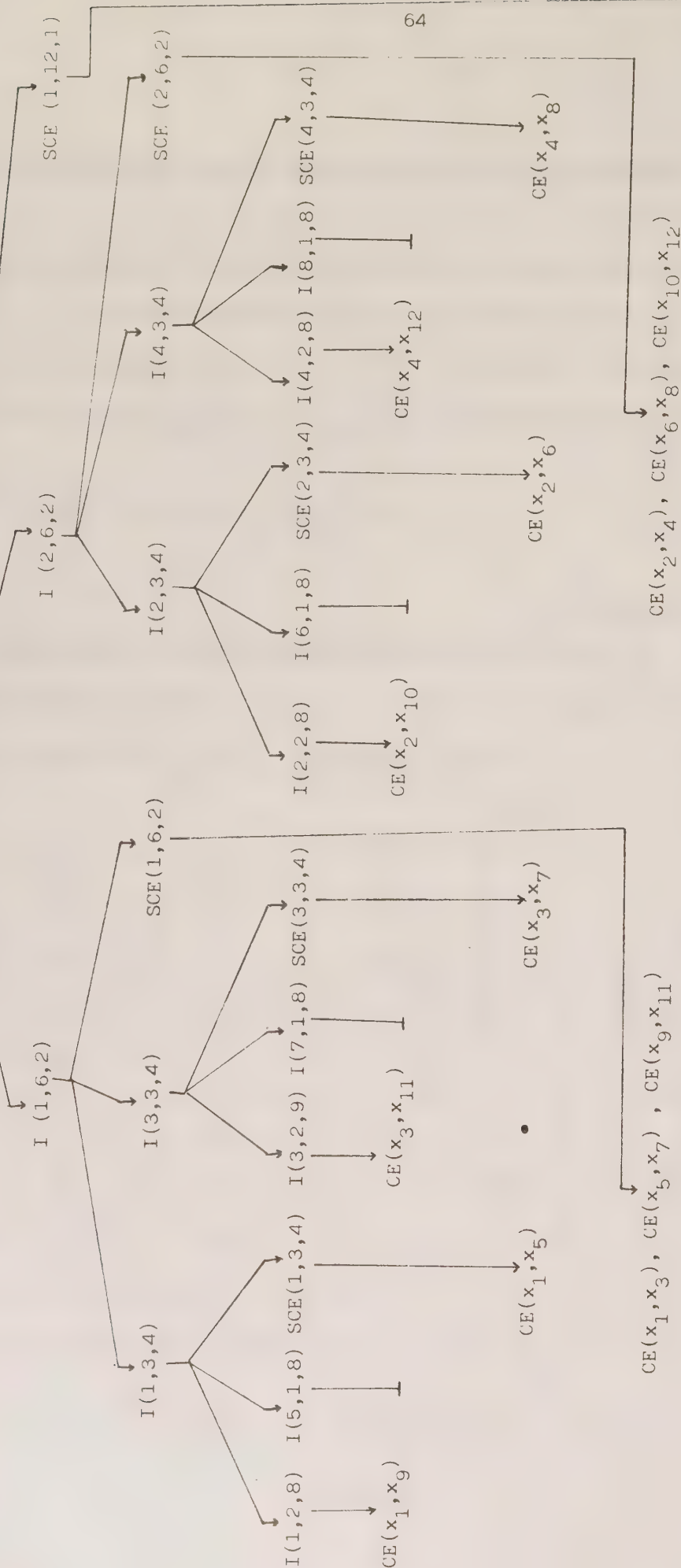
$x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 x_{10} x_{11} x_{12}$
 $I(1,12,1)$

 $CE(x_1, x_2), CE(x_3, x_4), CE(x_5, x_6), CE(x_7, x_8), CE(x_9, x_{10}), CE(x_{11}, x_{12})$

Figure 6

La procédure I est exécutée $\lceil \log_2 q \rceil$ fois, l'interclassement "bitonic" se fait donc en $\lceil \log_2 q \rceil$ unités de temps.

Soit $C(q)$ le nombre d'éléments de comparaison nécessaire pour faire l'interclassement. Il satisfait l'équation :

$$C(q) = C(\lceil q/2 \rceil) + C(\lfloor q/2 \rfloor) + \lfloor q/2 \rfloor \quad \text{pour } q > 2$$

et $C(2) = 1 \quad C(1) = 0$

Remarques:

* Le nombre d'éléments de comparaison correspond au nombre de comparaisons de l'algorithme dans le cas de l'exécution séquentielle.

* Si $q = 2^t$, $C(2^t) = t 2^{t-1}$ répartis en t niveaux c.à.d. dans chaque niveau, il y a 2^{t-1} éléments de comparaison. Dans ce cas, les deux méthodes d'interclassement (pair-impair et bitonic) possèdent le même nombre de niveaux, mais l'interclassement 'bitonic' compte un nombre d'éléments de comparaison supérieur à celui utilisé par l'autre méthode, ceci a l'avantage de construire un grand réseau de tri à partir d'un module standard de taille convenable.

(module est un nombre fixé d'éléments de comparaison).

Le tri par interclassement "bitonic" consiste à interclasser des monotonies de 2 éléments, puis de 4, etc... Dans le cas où q est une puissance de 2 ($q = 2^t$), le tri utilise $(t^2 + t) 2^{t-2}$ comparaisons en $\frac{t(t+1)}{2}$ niveaux.

4.4. Tri par Transposition

Son principe est d'appliquer l'opérateur comparaison-échange (CE) à une étape donnée, sur les indices impairs, puis sur les indices pairs à l'étape suivante. Ce procédé est répété un nombre de fois égal à la taille du fichier à trier.

Si n est le nombre d'éléments du fichier, on écrit alors l'algorithme suivant :

début

$j := 1$;

Tant que $j \leq n$ faire

début

pour $i := 1$ à $\lfloor \frac{n}{2} \rfloor$ faire $CE(x_{2i-1}, x_{2i})$ fait

$j := j+1$

Si $j \leq n$ alors (pour $i := 1$ à $\lceil \frac{n}{2} \rceil - 1$ faire $CE(x_{2i}, x_{2i+1})$ fait).

$j := j+1$) fsi

fin

fin

L'algorithme compte $\frac{1}{2} n (n-1)$ comparaisons qui peuvent s'exécuter en n unités de temps.

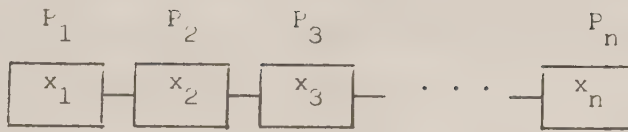
4.5. Exemple de Tri sur machine SIMD[23]

La performance des algorithmes de tri, pour des machines SIMD, dépend de la structure de la machine (c.à.d. de l'interconnexion des processeurs) et de l'algorithme utilisé. Le tri, sur ce type de machine, consiste à faire une numérotation sur les processeurs de façon à affecter au processeur n° i , le i^{e} plus petit article du fichier.

4.5.1. Interconnexion linéaire

On utilise la méthode de BATCHER qui sépare les indices impairs d'un côté et les indices pairs de l'autre; on interclasse les deux séparations indépendamment et on applique ensuite l'opérateur CE au résultat de cet interclassement. Pour cela, on suppose qu'on a deux monotonies à interclasser de longueur totale égale à n , chargées initialement dans n processeurs, l'une à côté de l'autre ; chaque processeur contient exactement un élément selon le

schéma suivant :



On note par :

$E(P_j, P_k)$: les processeurs de numéros j et k échangent leurs éléments

$CE(P_j, P_k)$: les processeurs de numéros j et k comparent leurs éléments (avec échange éventuel) de façon que P_j contienne le minimum de ces éléments et P_k contienne leur maximum ; pour $j < k$.

Algorithme 1:

étape 1 :

pour $i := 1$ à $\lfloor \frac{n-1}{2} \rfloor$ faire

(pour $j := i+1$ (pas 2) à $(n-i)$ faire $E(P_j, P_{j+1})$; fait) fait

étape 2 :

Appliquer l'algorithme Tri par transposition, en parallèle, sur les deux sous ensembles de processeurs :

$(P_1, P_2, \dots, P_{\lceil n/2 \rceil})$ et $(P_{\lceil n/2 \rceil + 1}, \dots, P_n)$

étape 3 :

pour $i := \lfloor \frac{n-1}{2} \rfloor$ (pas -1) à 1 faire

(pour $j := i+1$ (pas 2) à $(n-i)$ faire $E(P_j, P_{j+1})$; fait) fait

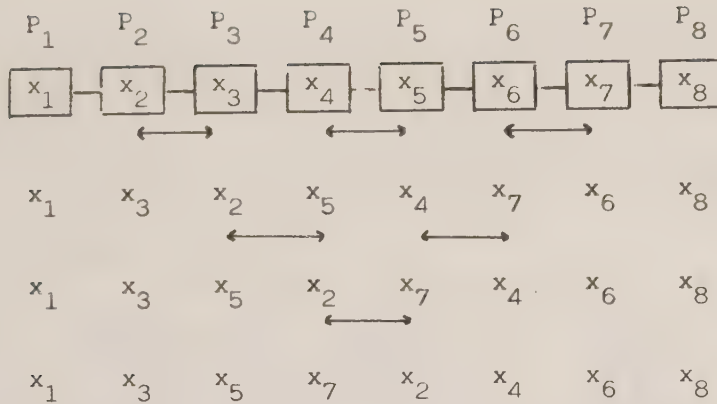
étape 4 :

pour $i := 2$ (pas 2) à $n-1$ faire $CE(P_i, P_{i+1})$; fait ;

Remarques :

L'étape 1 correspond à la séparation d'indices pairs et impairs.

exemple $n=8$: (" $\longleftrightarrow$ ") indique les échanges entre processeurs.



Les échanges de cette étape décrivent un triangle et s'exécutent en parallèle en $\lfloor \frac{n-1}{2} \rfloor$ unités de temps.

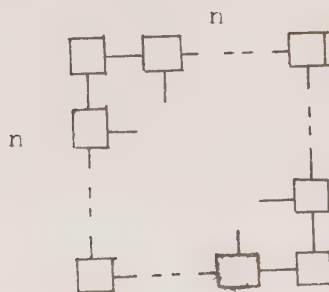
* L'étape 3 est l'inverse de l'étape 2 (triangle inversé). Elle est utile pour comparer deux éléments à distance d ; cette étape la ramène à 1. Les comparaisons se font pour les processeurs voisins. Le coût de cette étape est aussi $\lfloor \frac{n-1}{2} \rfloor$ unités de temps.

* Toutes les comparaisons-échanges de l'étape 4 se font en même temps.

* L'interclassement de deux monotonies de longueur totale n est donc en $O(n)$.

4.5.2. Interconnexion bidimensionnelle

Les processeurs sont disposés en forme d'une matrice selon le schéma suivant :



Le tri $N=n^2$ élément utilise l'algorithme de BATCHER énoncé précédemment : il est construit récursivement par un interclassement des monotonies rangées dans les processeurs colonnes de cette structure.

On suppose, pour simplifier, que les monotonies sont de longueurs égales et multiples de 2 (c.a.d. $n=2^q$). On définit une fonction γ d'indexation (ou numérotation) des processeurs de la façon suivante :

$$\begin{array}{ccc} \gamma : \{1,2, \dots, n\} \times \{1,2, \dots, m\} & \longrightarrow & \{1,2,3, \dots, nm\} \\ (i,j) & \longmapsto & \gamma_{ij} = \begin{cases} (i-1)m+j & \text{si } i \text{ est impair} \\ (i-1)m+m-j+1 & \text{si } i \text{ est pair} \end{cases} \end{array}$$

(cas d'une matrice de processeurs de n lignes et m colonnes)

Cette indexation est dite "row-major", elle signifie que les éléments seront triés de façon que le processeur indexé par γ_{ij} contiendrait le γ_{ij}^{e} plus petit du fichier pour tout i ($1 \leq i \leq n$) et pour tout j ($1 \leq j \leq n$). Son choix (parmi d'autres) est utile pour rendre l'algorithme utilisé plus performant. Soit $M(n,2)$ ($n > 1$) l'algorithme qui interclasse deux monotonies de longueurs égales. Chaque monotonie est rangée sur une colonne de processeurs :

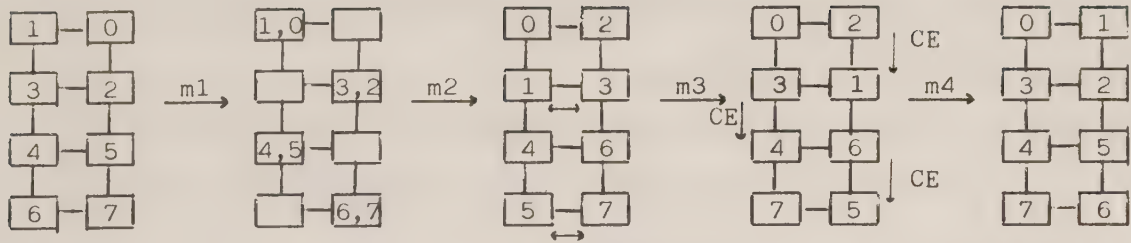
étape m1 :
mettre les éléments d'indices impairs de chaque colonne à gauche et ceux d'indices pairs à droite (voir exemple).

étape m2 :
trier chaque colonne par l'algorithme "tri par transposition".

étape m3 :
échanger les éléments de chaque ligne paire.

étape m4 :
appliquer l'opérateur CE sur les éléments des deux colonnes : on compare chaque processeur d'index pair avec le processeur d'index impair suivant.

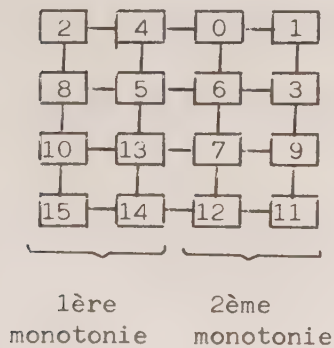
Exemple ($n=4$)



Soit $M(n,k)$ ($k > 2$) l'algorithme qui interclasse deux monotonies de longueur totale égale à $n \times k$ (la longueur de chaque monotonie est égale à $\frac{n \times k}{2}$). Chaque monotonie est chargée dans un sous ensemble de processeurs ($n \times k/2$) selon l'indexation des processeurs comme dans l'exemple suivant :

Si on a deux monotonies de 8 éléments chacune, avec $n=4$ ($k=4$) :

(2,4,5,8,10,13,14,15) et (0,1,3,6,7,9,11,12) . Elles seront chargées dans les processeurs de la façon suivante :



$M(n,k)$ est un algorithme récursif.

$M(n,k) =$

début si $k = 2$ alors ($M(n,2)$, return) fsi ;

étape M1 :

échanger les éléments de chaque ligne paire, ces éléments sont pris deux par deux.

étape M2 :

appliquer l'étape 1 de l'algorithme 1 (avec $n=k$) sur chaque ligne, considérée comme une interconnexion linéaire.

étape M3 :

lancer deux appels en même temps $M(n,k/2)$ sur les deux moitiés.

(Remarque : $M(n,k)$ est considérée comme une matrice de processeurs de n lignes et de k colonnes).

étape M4 :

appliquer l'étape 3 de l'algorithme 1 sur chaque ligne (avec $n=k$)

étape M5 :

échanger les éléments de chaque ligne paire ; ces éléments sont échangés deux par deux.

étape M6 :

appliquer les CE sur les processeurs indexés par γ_{ij}

pour tout i ($1 \leq i \leq n$) et pour tout j ($1 \leq j \leq k$).

On compare chaque processeur d'index pair à celui d'index impair suivant.

fin

Le tri d'un fichier de $n \times n$ éléments consiste, dans une première étape,

à trier n colonnes en parallèle et à appeler ensuite $M(n,2)$, puis $M(n,4), \dots, M(n,n)$.

CHAPITRE V

TRIS ASYNCHRONES

CHAPITRE V

TRIS ASYNCHRONES5.1. Introduction

Les structures MIMD imposent une conception algorithmique asynchrone. Chaque processeur exécute un traitement local et indépendant avec des communications éventuelles.

On distingue trois types de communications :

- (1) Arrivé à un point de son exécution, un processeur communique avec l'extérieur.
Ce type de communication est dit "couplage faible".
- (2) A chaque opération, les processeurs échangent des informations. Ils sont dits "fortement couplés".
- (3) Aucun échange ne se produit pendant l'exécution de l'algorithme; on dit alors que les processeurs sont "non couplés".

On trouve dans (TRE [38]) une classification d'algorithmes synchrones et asynchrones, et en particulier dans NAK [29], des algorithmes du type (1) et (3).

Une conception pour exprimer l'asynchronisme dans les programmes parallèles (PER [31]) consiste à décomposer un programme en "modules" (un module est défini comme une entité exécutable sur un processeur). Chaque module comporte un certain nombre de phases :

- traitements locaux,
- échange d'informations avec d'autres modules (avec attente éventuelle).

L'échange est exprimé par les "points d'intercommunication". Ceci est réalisé sur la décomposition d'un nombre en facteurs premiers (PER [32]).

Dans ce chapitre, nous nous intéressons aux algorithmes de tri, à leur transformation et à leur adaptation au calcul parallèle. Nous présentons deux algorithmes, le premier est un algorithme par interclassement. Nous n'avons pas pu tester ces algorithmes sur multiprocesseurs, toutefois nous en donnons une démonstration à l'aide des assertions que nous énonçons en 5.2. et nous faisons des simulations. Nous montrons ensuite l'application de l'interclassement au tri par sélection quadratique.

Le deuxième algorithme est celui du tri par comptage, qui consiste à rendre les processus les plus indépendants possible dans leurs exécutions ; le parallélisme est visible car il n'y a pas d'échange entre processeurs.

Nous donnons à la fin de ce chapitre, une liste d'algorithmes de tri "parallélisés", ainsi que leurs complexités.

5.2. Preuve de programmes

5.2.1. Axiomes de preuve de programmes séquentiels : (Hoa [19])

A chaque instruction (ou groupe d'instructions) (E) est associé un couple de prédicats (p,q) tel que :

Si p est vrai avant l'exécution de E alors q est vrai après cette exécution, et on note : $p \{E\} q$.

Nous donnons un bref rappel des règles de preuve en utilisant les notations suivantes :

- $\frac{E}{F}$ signifie que E implique F
- les énoncés sont délimités par des accolades.

1°) Règle conditionnelle :

$$\frac{p \text{ et } t \{E\} q, \quad p \text{ et non } t \{F\} q}{p \{ \text{si } t \text{ alors } E \text{ sinon } F \text{ fsi} \} q}$$

2°) Règle itérative :

$$\frac{p \text{ et } t \{E\} p}{p \{ \text{tant que } t \text{ faire } E \text{ fait} \} p \text{ et non } t}$$

3°) Règle de composition séquentielle :

$$\frac{p \{E\} q, q \{F\} r}{p \{E ; F\} r}$$

4°) Axiomes d'implication :

$$4.a \quad \frac{p \{E\} q, q \Rightarrow r}{p \{E\} r}$$

$$4.b \quad \frac{r \Rightarrow p, p \{E\} q}{r \{E\} q}$$

5.2.2. Preuve de programmes parallèles : (ow [30])

Un programme parallèle peut être défini comme un ensemble de processus $\{S_i\}$ pouvant être exécutés simultanément.

Donner une preuve d'un tel programme consiste à :

a) établir une preuve pour chaque processus S_i , considéré comme un programme séquentiel autonome ;

b) montrer que l'exécution de tout autre processus S_j n'interfère pas sur l'écriture des assertions de S_i

La non interférence peut être exprimée ainsi :

Soient S et T deux processus, P et Q deux prédicats; on dit que T n'interfère pas sur la preuve de $\{P\} S \{Q\}$ si et seulement si on vérifie les deux conditions suivantes :

1°) $\{Q \wedge \text{Apré}(T)\} T \{Q\}$ i.e: l'exécution de T ne remet pas en cause l'assertion finale Q.

2°) pour toute partie S' de S :

$$\{\text{pré}(S') \wedge \text{pré}(T)\} \text{ T } \{\text{pré}(S')\}$$

où $\text{pré}(A)$ est le prédicat avant l'exécution de A .

5.3. Tri par interclassement en parallèle

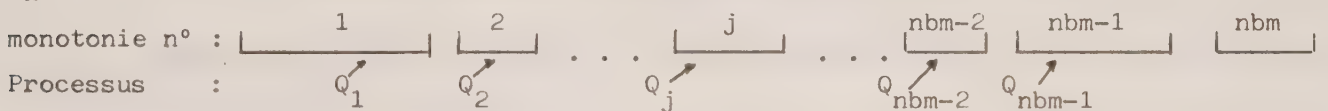
5.3.1. Principe de l'algorithme

Etant donné un tableau A , de n éléments à trier dans un certain ordre. Ce tableau est constitué des monotonies de longueur variable. Soit nbm , le nombre de ces monotonies. La "parallélisation" est basée sur l'idée suivante :

On lance $\text{nbm}-1$ processus en même temps. Chaque processus travaille sur une monotonie qu'il recopie dans une zone propre à lui et l'interclasse en fonction du résultat d'interclassement des autres processus commencés avant lui :

$\forall i \in \{1, 2, \dots, \text{nbm}-1\}$ tel que $j = \text{nbm}-i$, le processus numéro j , noté Q_j , travaille à partir de la monotonie numéro j , en fonction du résultat des processus

Q_k ($k > j$) selon le schéma suivant :



$Q_{\text{nbm}-1}$ commence à interclasser sa monotonie ($\text{nbm}-1$) avec la $\text{nbm}^{\text{ième}}$, dès qu'il place un élément, $Q_{\text{nbm}-2}$ se déclenche pour interclasser sa monotonie ($\text{nbm}-2$) avec les éléments produits par $Q_{\text{nbm}-1}$, au fur et à mesure de leur traitement (traitement signifie qu'on compare deux éléments et on place le plus petit, cet élément est dit "produit"). Il en est de même pour les autres processus. On définit donc un ordre sur ces processus de la façon suivante :

Si $k \geq l$ alors $Q_k \geq Q_l$ qui signifie que lorsque Q_k et Q_l sont actifs, Q_k est en avance sur Q_l .

Tout processus ne peut progresser dans son travail que lorsque son premier successeur l'avertit (sauf pour Q_{nbm-1} qui travaille sans se bloquer parcequ'il a la dernière monotonie produite).

Pour la réalisation de l'algorithme, nous définissons des tableaux pour contrôler les activités des processus :

- DEB(1:nbm) qui indique l'indice de début de chaque monotonie

- un tableau Y(1:nbm) initialisé ainsi :

$$\forall i \in \{1, 2, \dots, nbm-1\} ; Y[i] \leftarrow DEB[i] \text{ et } Y[nbm] \leftarrow n+1$$

Chaque fois un processus Q_j est actif, il incrémente sa composante en Y ($Y[j]$) de telle façon que : $Y[j] - DEB[j]$ indique, à tout instant, le nombre d'éléments produits par ce processus. On remarque que $Y[nbm] - DEB[nbm] = l_{nbm}$ où l_{nbm} est la longueur de la dernière monotonie.

De même, on utilise un tableau X(1:nbm), initialisé à 0, pour indiquer qu'un processus a terminé ou non ; lorsqu'un processus (Q_j) termine, il exécute $X[j] := 1$.

5.3.2. Programme

DEBUT

```

(* *** PREMIERE PHASE : CONSTITUTION DES MONOTONIES *** *)
NBM:=1; DEB [1] :=1;
POUR I:=1 HAUT N-1 FAIRE
  SI A [I+1]<A [I] ALORS
    DEBUT
      NBM:=NBM+1; DEB [NBM]:=I+1
    FIN;
(* ***** LES TABLEAUX A,DEB,Y ET X SONT COMMUNS A TOUS LES PROCESSES *** *)
(* INITIALISATION *)
POUR I:=1 HAUT NBM-1 FAIRE
  DEBUT X [I] :=0; Y [I] :=DEB [I] FIN;
  X [NBM] :=1; Y [NBM] :=N+1;
(* LANCEMENT DE L'EXECUTION PARALLELE *)
POUR I:=1 HAUT NBM-1 FAIRE EN PARALLELE
  DEBUT DU BLOC PARALLELE
    (* ***** TOUTES LES VARIABLES SUIVANTES SONT PROPRES A CHAQUE PROCESSUS *)
    (* *** D,F,DEB1,DEB2,FIN1,RG,J,JJ ET LE TABLEAU B *** *)
    (1) J:=NBM-I; DEB1:=DEB [J] ; DEB2:=DEB [J+1] ; FIN1:=DEB2-1; D:=0;
    POUR RG:=DEB1 HAUT FIN1 FAIRE
      A DEBUT
        D:=D+1; B [D] :=A [RG]
      BLOC A
        FIN; (*Chaque processeur recopie une monotonie dans une zone "B"
              propre à lui *)
    (2) F:=D; D:=1; JJ:=J;
    (* MODULE B *)
    (3) TANTQUE ( D <= F ) ET ( DEB2 <= N ) FAIRE
      DEBUT
        (4) TANTQUE ( DEB2 >= Y [JJ+1] ) ET ( X [JJ+1] =1 ) FAIRE JJ:=JJ+1;
        (5) SI DEB2 < Y [JJ+1] ALORS
          DEBUT
            RG:=DEB2;
            (6) TANTQUE ( RG=DEB2 ) ET ( D <= F ) FAIRE
              DEBUT
                SI B [D] > A [DEB2] ALORS
                  DEBUT
                    A [DEB1] :=A [DEB2] ; DEB2:=DEB2+1;
                    DEB1:=DEB1+1; Y [J] :=DEB1
                  FIN
                SINON
                  DEBUT
                    A [DEB1] :=B [D] ; D:=D+1;
                    DEB1:=DEB1+1; Y [J] :=DEB1
                  FIN
              FIN tant que
            FIN si
          FIN tant que
        FIN tant que
      FIN tant que
    FIN tant que
  FIN tant que

```

```
|  
|  
|  
| (* MODULE C *)  
|  
| (7) TANTQUE D <=F FAIRE  
|   DEBUT  
|     A [DEB1] := B [D] ; D := D+1  
|     DEB1 := DEB1+1 ; Y [j] := DEB1  
|   FIN;  
| (8) X [J] := 1  
|  
| FIN DU BLOC PARALLELE
```

FIN DU PROGRAMME.

5.3.3. Commentaires sur le programme

C1. première phase : on parcourt le tableau A pour déterminer les monotonies et marquer leurs indices dans le tableau DEB.

C2. phase d'initialisation :

on initialise les tableaux X et Y qui sont partagés par les processus, ainsi que les tableaux A et DEB.

C3. Lancement des processus en parallèle :

les processus subissent le même traitement ; mais pas à tout instant. C'est donc un processus itératif. On a choisi l'instruction : pour $I:=1$ Haut $\text{nbm}-1$ faire en parallèle qui est du même type que "Avec" du pastoral (DEMAS [26]).
- l'index du processus est l'identificateur j où $j=\text{nbm}-I$.

Chaque processus possède des variables propres à lui :

$D, F, \text{DEB}_1, \text{FIN}_1, \text{RG}, J, \text{JJ}$ et un tableau B.

Pour qu'il n'y ait pas d'ambiguïté, on notera par x^j , la variable x propriété du processus Q_j .

C3.1. Premier travail à faire par chaque processus lancé :

chaque processus Q_j recopie la $j^{\text{ème}}$ monotonie dans une zone propre à lui, notée B^j , à partir de l'indice $\text{DEB}[j]$ jusque $\text{DEB}[j+1]-1$, indépendamment des autres. C'est le module A.

C3.2. Module B: (bloc du tant que (3))

Dans chaque processus Q_j , on cherche à interclasser les éléments de sa monotonie qui sont dans B^j , avec une autre monotonie dans le tableau A, à partir de l'indice $\text{DEB}[j+1]$; on range, après comparaison, le plus petit dans le tableau commun A, à partir de l'indice $\text{DEB}[j]$. On utilise donc

les variables suivantes pour chaque processus : D^j parcourt le tableau B^j ,

deb_2^j parcourt la 2ème monotonie dans A , qui résulte de l'interclassement des autres processus.

deb_1^j : indice de rangement dans A , du processus Q_j

Ceci est réalisé selon le schéma suivant :

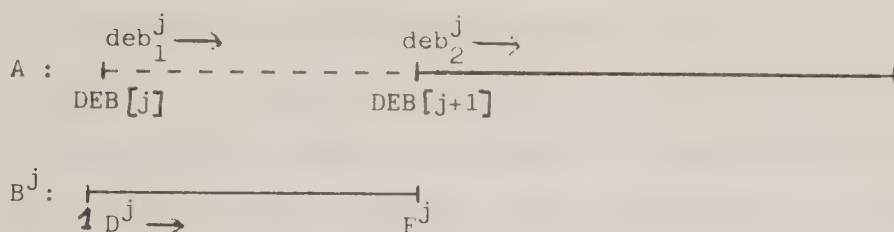


Figure 2

C3.2.1. Synchronisation

Supposons que le processus Q_j soit lié à l'avancement de Q_k ($k > j$) avec Q_j et Q_k actifs, la condition d'avancement de Q_j est : $deb_2^j < deb_j^k$. En effet, au départ, $k=j+1$ et $deb_2^j = DEB[j+1] = deb_1^{j+1}$ (la première égalité est due à (1) du programme); ceci étant vérifié pour tous les processus, sauf pour Q_{nbm-1} (c.à.d. $j=nbm-1$) qui a $deb_2^{nbm-1} = DEB[nbm] < n+1 = Y[j+1]$. Cette condition d'avancement est exprimée dans le programme, par le test (5). On remarque que $Y[j+1]$ n'est rien d'autre que deb_1^{j+1} (voir preuve du programme) en convenant que $deb_1^{nbm} = Y[nbm]$.

La synchronisation est exprimée ainsi : un processus, qui ne vérifie pas le test (5), boucle sur l'instruction "tant que (3) (sans rien faire) tout en regardant le processus auquel il est lié, s'il a incrémenté sa composante en Y (c.à.d. s'il a produit un élément) ou s'il a terminé (sa composante en X égale à 1) et tous ses éléments produits sont traités ((4) et (5) du programme).

Remarque : On aurait pu supprimer le tableau Y et le remplacer par un tableau DEB1, mais la présentation en aurait été alourdie.

C3.3. Module C (tant que (7))

Quand un processus Q_j termine son exécution en module B, il sort du "tant que (3)" soit par $D^j > F^j$, soit par $deb_2^j > N$. S'il en sort par $D^j > F^j$, il a donc terminé (i.e. rangé tous ses éléments dans le tableau A), il n'exécute pas le module C et termine par exécuter l'instruction (8) pour indiquer aux autres processus qu'il a terminé. Par contre, s'il en sort avec $D^j \leq F^j$, il boucle sur "tant que (7)" jusqu'à ce qu'il range tous ses éléments, restant dans son tampon B^j . Tout cela est sans interférence avec les autres processus qu'on montrera plus loin.

(on montrera qu'un seul processus, au plus, pourrait se trouver en C à la fois).

Dans tous les cas, on termine par affecter à $X[j]$ la valeur 1, pour tout processus Q_j (instruction (8)).

5.3.4. Preuve du programme

Plan de la démonstration :

I. module A : Quand un processus recopie sa monotonie, cela se passe sans interférence avec les autres processus parallèles.

II. modules B et C (car C dépend de la terminaison de B) : On définira un prédicat P_j et on montrera son invariance par l'exécution de Q_j et par tous les processus parallèles à Q_j .

III. Terminaison des processus : lorsque tous les processus terminent, le tableau A sera trié.

Démonstration :

I. Module A :

Dans chaque processus Q_i , on recopie la monotonie n°i de l'indice $DEB[i]$ à $DEB[i+1] - 1$. Or les ensembles $I_i = \{ DEB[i], DEB[i] + 1, \dots, DEB[i+1] - 1 \} \forall 1 \leq i \leq nbm-1$ sont disjoints, c.à.d. $\forall 1 \leq i \leq nbm-1, \forall 1 \leq j \leq nbm-1$ avec $j \neq i$ on a $I_i \cap I_j = \emptyset$ du fait que $DEB[k] < DEB[k+1] \forall 1 \leq k \leq nbm-1$.

Dans chaque processus on fait : (notation : " $\leftarrow$ " = affectation)
 $\forall k^i \in I_i = \{ DEB[i], \dots, DEB[i+1] - 1 \}$ t.q. $D^i = k^i - DEB[i] + 1$ on a $(B[D])^i \leftarrow A[k^i]$
 et quand un processus termine de recopier, on aura :

$$k^i = DEB[i+1] - 1 \quad D^i = DEB[i+1] - DEB[i] \longrightarrow F^i$$

(α) : $F^i = DEB[i+1] - DEB[i]$ indique la longueur de la monotonie i, et

donc ce module est sans interférence entre processus.

On ne fait aucune supposition sur la vitesse de traitement des processeurs.

Si dans un processus Q_i , on recopie plus vite que les autres, il se bloque dans le module B (bloc B_1) parceque le test (5) n'est pas vérifié (du fait de l'initialisation de Y et $deb_2^i = DEB[i+1] = Y[i+1]$) sauf pour Q_{nbm-1} qui doit commencer le premier et que tous les autres processus doivent attendre qu'il commence.

II. On définit un prédicat P_j associé au processus Q_j de la façon suivante :

P_j : Pour tout deb_2^j compris entre $DEB[j+1]$ et n,

Pour tout D^j compris entre 1 et F^j

1) on a $deb_1^j = deb_2^j - F^j + D^j - 1$

2) pour tout k compris entre $DEB[j]$ et deb_1^j , on a $A[deb_1^j] \geq A[k]$

Avant de démontrer son invariance, on démontre d'abord quelques relations

invariantes par l'exécution de Q_j et par l'exécution du processus auquel Q_j

est lié :

$$R1 : deb_2^j - deb_1^j = F^j - D^{j+1} ; Y[j] = deb_1^j$$

$$R2 : deb_1^j < deb_2^j$$

R3 : il existe $JJ \geq J$ pour lequel on a :

$$R3.1 \quad deb_2^j \leq Y[jj+1]$$

$$R3.2 \quad deb_1^j < deb_1^{jj+1}$$

Remarque : ces relations sont définies pour tout processus actif, pour lequel on veut démontrer que son exécution est valide.

Signification de ces relations :

R1 : (voir le schéma de la figure 2 (page 79))

On range le résultat de l'interclassement dans le tableau A à l'indice deb_1^j , chaque élément rangé à cet indice est pris, soit du tableau B^j à l'indice D^j , soit du tableau A lui-même, mais à l'indice deb_2^j . R1 signifie donc, qu'on peut ranger les éléments de B^j sans perdre aucune information du tableau A.

R2 est vérifié pour tout processus actif, et lorsque $deb_1 = deb_2$, le processus terminera son exécution.

R3.1 est lié à la condition d'avancement ou d'attente du processus Q_j

R3.2 : deux processus différents ne peuvent ranger leurs éléments dans une même zone.

Preuve de R1 :

Au départ (avant l'exécution du module B) on a :

$$deb_2^j = DEB[j+1] , \quad deb_1^j = DEB[j] \text{ (ligne (1) du programme) ; } D^j=1 \text{ (ligne (2))}$$

et d'après (α) de la page 81, on a : $F^j = \text{DEB}[j+1] - \text{DEB}[j]$ et à l'initialisation $Y[j] = \text{DEB}[j] \quad \forall 1 \leq j \leq \text{nbm}-1$.

Donc R1 est vérifiée au départ.

* invariance de R1 par l'exécution du module B.

Supposons que les "tant que" de ce module terminent, alors on a :

$$\begin{aligned} & \left(\text{deb}_2^j - \text{deb}_1^j = F^j - D^j + 1; \text{deb}_1^j = Y[j] \right) \underline{\text{et}} \left((B[D])^j > A[\text{deb}_2^j] \right) \left\{ \text{deb}_2^j := \text{deb}_2^j + 1; \text{deb}_1^j := \text{deb}_1^j + 1; \right. \\ & \quad \left. Y[j] := \text{deb}_1^j \right\} \left((\text{deb}_2^j - 1) - (\text{deb}_1^j - 1) = F^j - D^j + 1; Y[j] = \text{deb}_1^j \right) = \left(\text{deb}_2^j - \text{deb}_1^j = F^j - D^j + 1; Y[j] = \text{deb}_1^j \right) \\ & \left(\text{deb}_2^j - \text{deb}_1^j = F^j - D^j + 1; \text{deb}_1^j = Y[j] \right) \underline{\text{et}} \left((B[D])^j \leq A[\text{deb}_2^j] \right) \left\{ D^j := D^j + 1; \text{deb}_1^j := \text{deb}_1^j + 1; Y[j] := \text{deb}_1^j \right\} \\ & \left(\text{deb}_2^j - (\text{deb}_1^j - 1) = F^j - (D^j - 1) + 1; Y[j] = \text{deb}_1^j \right) = \left(\text{deb}_2^j - \text{deb}_1^j = F^j - D^j + 1; Y[j] = \text{deb}_1^j \right) \end{aligned}$$

et d'après la règle d'interférence conditionnelle (règle 1°) du 5.2.)

avec

$p = q = R1$, $t = (B[D])^j > A[\text{deb}_2^j]$, $E = \text{corps de Si } t$, $F = \text{corps de Sinon de } t$
on a l'invariance de R1 pour le module B (par la règle itérative, règle 2°).

* Invariance de R1 par l'exécution du module C :

Notons que si le module B termine par $(D > F)$, le module C n'est pas exécuté et donc, pour qu'il soit exécuté, il faut que B termine par $D \leq F$ et $\text{deb}_2 > n$, puisque deb_2 est un entier, incrémenté à chaque fois de 1, alors $(\text{deb}_2 > n)$ sera traduit par $\text{deb}_2 = n+1$ et donc la relation R1 devient :

$n - \text{deb}_1^j = F^j - D^j$ ce qui signifie qu'il reste de la place, dans A, juste pour ranger les éléments de la monotonie $n = J$ restant dans B^j (de D^j à F^j) ;

L'invariant R1 reste donc vrai avant l'exécution du module C.

Invariance de R1 par le tant que (7) du module C. En appliquant la règle 2 du 5.2., on aura :

$$\begin{aligned} & \left(\text{deb}_2^j - \text{deb}_1^j = F^j - D^j + 1; \text{deb}_1^j = Y[j] \right) \underline{\text{et}} (D \leq F) \left\{ D^j := D^j + 1; \text{deb}_1^j := \text{deb}_1^j + 1; Y[j] := \text{deb}_1^j \right\} \\ & \left(\text{deb}_2^j - (\text{deb}_1^j - 1) = F^j - (D^j - 1) + 1; \text{deb}_1^j = Y[j] \right) = \left(\text{deb}_2^j - \text{deb}_1^j = F^j - D^j + 1; \text{deb}_1^j = Y[j] \right) \end{aligned}$$

D'où l'invariance de R1 par l'exécution de Q_j .

Preuve de R2

R2 est une conséquence immédiate de R1. En effet, on a $D^j \leq F^j$ pour tout processus actif et donc $F^j - D^j \geq 0$. D'après R1 on a :

$$\text{deb}_2^j - \text{deb}_1^j \geq 1 \implies \text{deb}_2^j > \text{deb}_1^j \quad \text{d'où R2.}$$

Preuve de R3

La suite de la démonstration est basée sur ceci :

Soit Q_k le processus auquel Q_j est lié avec $(k > j)$ t.q :

S₁ : * Q_j est actif

S₂ : * - ou bien Q_k n'a pas encore terminé

- ou bien Q_k a terminé, mais Q_j n'a pas traité tous les éléments produits par Q_k . (k correspond à $jj+1$ dans le programme, instructions (4) et (5))

En ce qui concerne la dernière monotonie nbm, on suppose qu'elle a été produite par un processus Q_{nbm} avec $X[nbm]=1$, ceci nous assure l'existence de jj de R3.

Admettons qu'il existe p processeurs avec $P \geq nbm-1$ (si $P < nbm-1$, la démonstration se déduit facilement de celle-ci).

Preuve de R3.1 ($\text{deb}_2^j \leq Y[jj+1]$)

Au départ, tous les processus sont actifs, donc $jj=j$ et on a :

$$\forall j ; 1 \leq j \leq nbm-1 \quad \text{deb}_2^j = \text{DEB}[j+1] = Y[jj+1] \quad (\text{voir Initialisation})$$

pour que $\text{deb}_2^j < Y[jj+1]$, il faut avoir incrémenté $Y[jj+1]$ dans le processus

Q_{jj+1} ; puisque les processus sont dépendants (Q_i dépend de $Q_{i+1} \quad \forall i \geq 1$),

il suffit donc que Q_{nbm-1} (qui commence le premier) ne se bloque pas par la

condition (5) du programme et incrémente, à chaque passage, $Y[nbm-1]$ et que tout autre processus incrémente sa composante en Y . En effet, à l'initialisation : $deb_2^{nbm-1} = DEB[nbm] < n+1 = Y[nbm]$, et d'après la condition du "tant que (3)" deb_2 peut aller jusqu'à n ; le processus Q_{nbm-1} ne peut se bloquer par la condition (5). Montrons que chaque processus Q_j ($1 \leq j \leq nbm-1$) incrémente $Y[j]$, une fois passé par le test $deb_2^j < Y[jj+1]$. (Remarquons que, d'après R1, on a $deb_1^j = Y[j]$ et l'écriture suivante :

$$\begin{aligned} & \{ deb_1^j := deb_1^j + 1, Y[j] := deb_1^j \} \text{ est équivalente à } \{ deb_1^j := deb_1^j + 1; Y[j] := Y[j+1] \} \\ & (deb_2^j < Y[jj+1]) \text{ et } ((B[D])^j > A[deb_2^j]) \{ deb_2^j := deb_2^j + 1; Y[j] := Y[j+1] \} (deb_2^j \leq Y[jj+1]) \\ (\mathcal{P}) & (deb_2^j < Y[jj+1]) \text{ et } ((B[D])^j \leq A[deb_2^j]) \{ Y[j] := Y[j+1] \} (deb_2^j < Y[jj+1]). \end{aligned}$$

Dans ce dernier schéma, on n'incrémente pas deb_2^j ; ce qui signifie qu'on boucle sur le bloc B2. Or, on a $(deb_2^j < Y[jj+1]) \Rightarrow (deb_2^j \leq Y[jj+1])$; d'après la règle d'implication (4.a), $(\mathcal{P})$ devient alors :

$$(deb_2^j \leq Y[jj+1]) \text{ et } ((B[D])^j \leq A[deb_2^j]) \{ Y[j] := Y[j+1] \} (deb_2^j \leq Y[jj+1])$$

On incrémente dans tous les cas, $Y[j]$. Par application de la règle conditionnelle 3°) on aura l'invariance de R3.1.

Remarque : L'indice jj du "tant que (4)" ne peut dépasser $nbm-1$, en effet pour $jj = nbm-1$ on a $Y[jj+1] = n+1$ et $deb_2 \leq n$ (condition du "tant que (3)"); jj ne peut pas être incrémenté car la condition $(deb_2 \geq Y[jj+1])$ du "tant que (4)" n'est pas vérifiée; elle le serait dans le cas où $deb_2 \geq n+1$. Mais dans ce cas, on sortira de la boucle du "tant que (3)".

Preuve de R.3.2. : $(deb_1^j < deb_1^{jj+1})$

D'après R1 on a $deb_1^i = Y[i] \quad \forall 1 \leq i \leq nbm-1$; R3.1 qui est $deb_2^j \leq Y[jj+1]$, est équivalente à $deb_2^j \leq deb_1^{jj+1}$ (en admettant que $deb_1^{nbm} = Y[nbm]$). Par cette relation et celle de R2 (qui est $deb_1^j < deb_2^j$), on a :

$$deb_1^j < deb_2^j \leq deb_1^{jj+1} \Rightarrow deb_1^j < deb_1^{jj+1} \quad \text{et donc R.3.2.}$$

Remarques et conséquences :

Les règles R3.1 et R3.2 peuvent être énoncées ainsi :

il existe k ($k > j$), pour lequel Q_k est le premier processus, actif, voisin de Q_j vérifiant S2 (de la page 84) tel qu'on a :

$$R3.1 : \text{deb}_2^j \leq \text{deb}_1^k = Y[k] \quad R3.2 : \text{deb}_1^j < \text{deb}_1^k .$$

On démontre par récurrence que : pour tout j compris entre 1 et $\text{nbm}-1$ et pour tout ℓ compris entre j et nbm pour lesquels Q_j et Q_ℓ sont actifs, on a :

$$R'3.1 : \text{deb}_2^j \leq \text{deb}_1^\ell$$

$$R'3.2 : \text{deb}_1^j < \text{deb}_1^\ell$$

R4 : un seul processus peut se trouver en module C à la fois

Remarque : Tout processus Q_k termine par $D^k > F^k$ (i.e. $F^k = D^{k+1}$), c'est-à-dire qu'il n'a plus d'éléments à ranger.

En remplaçant F^k , dans la relation R1, par D^{k+1} , on obtient

$$\text{deb}_2^k = \text{deb}_1^k$$
Démonstration :

Le module C ne peut être exécuté, par un processus Q_j , que lorsque $D^j \leq F^j$ (en sortant du module B par $\text{deb}_2^j > n$, i.e. $\text{deb}_2^j = n+1$).

Soit Q_i un processus actif lié à Q_j avec $i < j$ alors :

R'4 : Q_i ne peut se trouver en module C avant que Q_j termine : En effet,

supposons que Q_j s'exécute en C et n'a pas encore terminé, il a donc $D^j \leq F^j$ et $\text{deb}_2^j = n+1$. Dire que Q_i est actif est équivalent à dire que $D^i \leq F^i$.

On a $\text{deb}_2^i \leq \text{deb}_1^j$ (d'après R'3.1) et $\text{deb}_1^j < \text{deb}_2^j$ (d'après R2), ce qui nous donne :

$$\text{deb}_2^i \leq \text{deb}_1^j < \text{deb}_2^j = n+1 \implies \text{deb}_2^i < n+1 \implies \text{deb}_2^i < n$$

Le processus Q_i a : $D^i \leq F^i$ et $\text{deb}_2^i \leq n$, il ne peut donc sortir du module B car la condition du "tant que (3)" est encore vérifiée.

Par récurrence sur j on démontre (R'4) $\forall 1 \leq j \leq nbm-1$, $\forall 1 \leq i \leq j$ et par conséquent R4.

Remarque : on utilise la récurrence parce que les processus sont liés dans

leurs exécutions en ce sens que Q_{i+1} donne du travail à Q_i $\forall 1 \leq i \leq nbm-1$

Invariance de P_j par l'exécution des processus parallèles à Q_j :

Il suffit de le démontrer pour un processus Q_i ($i < j$) qui dépend de Q_j et par récurrence sur j , il sera généralisé.

Remarquons d'abord que la première égalité de P_j ($\text{deb}_1^j = \text{deb}_2^j - F^j + D^j - 1$) est obtenue de la relation invariante R1.

Posons $k^j = \text{DEB}[j]$ et $m^i = \text{DEB}[i]$; $m^i < k^j$ car $i < j$. P_j se traduit par $\{A[k^j], A[k^j+1], \dots, A[\text{deb}_1^j]\}$ trié dans l'ordre croissant (ce sont les éléments de Q_j). De même Q_i a $\{A[m^i], A[m^i+1], \dots, A[\text{deb}_1^i]\}$ trié.

$i < j$ entraîne $\text{deb}_1^i < \text{deb}_1^j$ (d'après R'3.2), on distingue deux cas :

a) $\text{deb}_1^i < \text{DEB}[j]$, dans ce cas les éléments de Q_j restent inchangés.

b) $\text{DEB}[j] < \text{deb}_1^i < \text{deb}_1^j$.

D'après 2) de P_j :

$\forall s$ ($\text{DEB}[j] < s < \text{deb}_1^j$) on a $A[\text{deb}_1^j] \geq A[s]$; les éléments de Q_j restent

donc triés et par conséquent, on a :

$\{A[m^i], A[m^i+1], \dots, A[\text{deb}_1^j]\}$ trié.

Conséquence :

Lorsque les processus terminent, si on prend $i=1$, on a $m^1 = \text{DEB}[1] = 1$ et pour $j = \text{nbm}-1$, $\text{deb}_2^{\text{nbm}-1}$ peut aller jusque n selon la nature des éléments de la monotonie $\text{nbm}-1$, on a, d'après la remarque de la page 86, $\text{deb}_1^{\text{nbm}-1} = \text{deb}_2^{\text{nbm}-1}$ et par conséquent :

$$\left\{ A[1], A[2], \dots, A[\text{deb}_2^{\text{nbm}-1}-1] \right\} \text{ trié}$$

Si $\text{deb}_2^{\text{nbm}-1} = n+1$ (dû au "tant que (3)") alors le tableau A sera trié ;

le cas contraire ($\text{deb}_2^{\text{nbm}-1} \leq n$) signifie que les éléments maximaux du tableau A sont contenus dans la dernière monotonie. Ils gardent donc leurs bonnes places.

Il ne reste que la démonstration de l'invariance de P_j par l'exécution de Q_j et l'arrêt des processus pour achever la démonstration du programme.

Invariance de P_j par l'exécution de Q_j

La démonstration est classique car l'exécution de chaque processus est séquentielle. Toutefois nous en donnons des remarques :

- P_j est vrai au départ ; en effet $D^j = 1$, $\text{deb}_2^j = \text{DEB}[j+1]$ et $F^j = \text{DEB}[j+1] - \text{DEB}[j]$ donc, $\text{deb}_1^j = \text{deb}_2^j - F^j + D^j - 1 = \text{DEB}[j]$ et $A[\text{deb}_1^j] \geq A[\text{deb}_1^j]$ qui est vrai.
 - L'exécution de Q_j consiste à interclasser la monotonie $(B(k))_{k=1, F=\text{DEB}[j+1]-\text{DEB}[j]}$ avec une autre monotonie $(A(l))_{l=\text{DEB}[j+1], n}$, le résultat d'interclassement sera rangé dans le tableau A à partir de l'indice $\text{DEB}[j]$. Il est facile de voir qu'à chaque exécution du programme, si on pose $m = \text{DEB}[j]$, on a :
- $$\left\{ A[m], A[m+1], \dots, A[\text{deb}_1^j] \right\} \text{ trié.}$$

III. Terminaison des processus

* Terminaison du module A : évident car le corps de "Pour" est contrôlé par un entier positif RG qui est borné.

* Terminaison du module B :

Le "tant que (3)" est contrôlé par deux entiers D et deb_2 bornés (D par F et deb_2 par n). deb_2 est incrémenté dans la vérité du test ($B[D] > A[deb_2]$) et D dans le cas de sinon. Le raisonnement est le même pour les autres "tant que".

On a vu (remarque de la page 85) que jj du "tant que (4)" ne peut dépasser nbm-1.

* Module C : idem.

Remarque : un processus Q_i ne peut se bloquer indéfiniment en bouclant sur "tant que (3)" ; en effet, si l'avancement de Q_i dépend de celui de Q_j ($j > i$), Q_i attend parce que le test (5), qui est $deb_2^i < Y[j]$, n'est pas vérifié ; d'après la preuve de R3, on a l'invariance de $deb_2^i \leq deb_1^j = Y[j]$; donc le test (5) n'est pas vérifié est équivalent à $deb_2^i = Y[j]$. On distingue deux cas :

a) ou bien Q_j est encore actif ; dans ce cas Q_j incrémente $Y[j]$ (voir preuve de programme).

b) ou bien Q_j a terminé ; dans ce cas Q_i boucle sur "tant que (4)" pour chercher un processus Q_k ($k > j > i$) tel que :

- ou bien Q_k est actif

- ou bien Q_k a terminé, mais Q_i n'a pas comparé tous les éléments produits par Q_k .

Si $k = nbm$, Q_i peut terminer son exécution avec $deb_2^i > n$ car $Y[nbm] = n+1$.

Remarques :

- Cet algorithme est asynchrone en ce sens que plusieurs processus peuvent s'exécuter en module A, en même temps avec l'exécution de B par un (ou plusieurs) processus et un en module C au plus.

- Activation des processus : cela pourrait être assuré par une instruction du type "Avec" du pastoral [26]. Le parallélisme, dans cet algorithme, peut être exprimé par la primitive NFØRK.

5.4. Estimation de la Complexité

Etant donné nbm , le nombre de monotonies décomposé en $(nbm-1)$ processus, on procède par tranches de p processeurs ; on suppose que $nbm-1$ est un multiple de p , c.à.d. il existe p, t, q :

$$nbm-1 = Bp \quad (1)$$

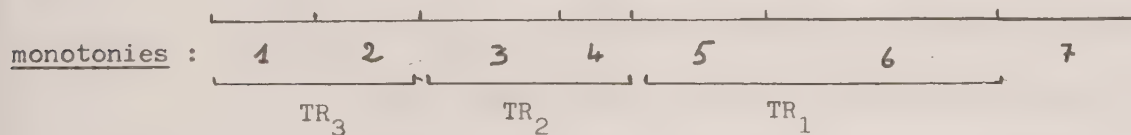
5.4.1. Complexité d'une tranche k

Dans une telle tranche, on interclasse p monotonies sur p processeurs, soit m_k le nombre d'opérations dans la tranche k (TR_k) ; une opération est définie comme étant une comparaison d'un élément de B à un autre de A et le rangement du plus petit ;

$$m_k = \max_{i \in TR_k} (nep_i + \alpha_i) \quad \text{avec} \quad TR_k = [nbm-kp, nbm - (k-1)p-1]$$

où :

- i est le numéro de monotonie sur laquelle travaille un processeur
- nep_i = nombre d'éléments produits par le processeur qui interclasse la monotonie i
- α_i = nombre de cycles d'attente du processeur qui essaie d'interclasser la monotonie i (un cycle correspond à une opération)
- TR_k est la tranche k , exprimée en nombre de monotonies; par exemple si on a $nbm = 7$ et $p = 2$: les monotonies sont numérotées comme suit :



Pour estimer α_i , on suppose qu'avant d'entamer la tranche $(k+1)$, tous les processeurs actifs dans la tranche k terminent leurs exécutions, c.à.d.

$0 \leq \alpha_i \leq p-1$ pour toutes les tranches : α_i est déterminé par la formule suivante :

$$\alpha_i = nbm - (k-1)p - i - 1 \quad \text{dans le sens suivant :}$$

le premier processeur de chaque tranche attend 0 cycle, le suivant 1 cycle ainsi de suite jusqu'au dernier de cette tranche qui attend $(p-1)$ cycles.

- Soit ℓ_i , la longueur de la monotonie i ; le processeur qui travaille sur la monotonie i produit, au maximum, $n - \sum_{j < i} \ell_j$ selon la nature des éléments qu'il a dans sa monotonie et donc $nep_i \leq n - \sum_{j < i} \ell_j$. On remplace α_i et nep_i dans m_k , par leurs valeurs respectives :

$$\begin{aligned} m_k &\leq \max_{i \in TR_k} \left[(n + nbm - kp + p - 1) - (i + \sum_{j < i} \ell_j) \right] \\ &\leq (n + nbm - kp + p - 1) + \max_{i \in TR_k} \left[- (i + \sum_{j < i} \ell_j) \right] \\ &\leq (n + nbm - kp - p - 1) - \min_{i \in TR_k} (i + \sum_{j < i} \ell_j) \end{aligned}$$

Le minimum de la quantité $i + \sum_{j < i} \ell_j$ s'obtient pour i égal à la borne inférieure de l'intervalle $TR_k = [nbm - kp, nbm - (k-1)p - 1]$ c.à.d. pour $i = nbm - kp$ et donc :

$$\begin{aligned} m_k &\leq (n + nbm - kp - p - 1) - (nbm - kp + \sum_{j < nbm - kp} \ell_j) \\ m_k &\leq n + p - 1 - \sum_{j < nbm - kp} \ell_j \end{aligned}$$

Posons $C_k = n + p - 1 - \sum_{j < nbm - kp} \ell_j$

Ceci étant fait pour une seule tranche ; or on a $\beta (= \frac{nbm-1}{p})$ tranches

d'après (1). La complexité globale est donc égale à $\sum_{k=1}^{\beta} m_k \leq \sum_{k=1}^{\beta} C_k$

Soit

$$C = \sum_{k=1}^{\beta} C_k = \sum_{k=1}^{\beta} (n + p - 1) - \sum_{k=1}^{\beta} \sum_{j < nbm - kp} \ell_j$$

$$C = \beta (n+p-1) - \sum_{k=1}^{\beta} \sum_{j < nbm-kp} \ell_j \quad (2).$$

Posons $S_k = nbm - kp$, $V_k = \sum_{j < S_k} \ell_j$ et $U_k = \sum_{j=(k-1)p+1}^{kp} \ell_j$ avec $U_0 = 0$

U_k = la somme des longueurs des monotones se trouvant dans la tranche k

Analysons $\sum_{k=1}^{\beta} V_k = \sum_{k=1}^{\beta} \sum_{j < nbm-kp} \ell_j$ de la formule (2).

pour $k = \beta = \frac{nbm-1}{p}$ on a $S_{\beta} = 1$ et $V_{\beta} = 0 = U_{\beta}$

$k = \beta - 1$ $S_{\beta-1} = p+1$ $V_{\beta-1} = \sum_{j=1}^p \ell_j = U_1$

$k = \beta - 2$ $S_{\beta-2} = 2p+1$ $V_{\beta-2} = \sum_{j=1}^{2p} \ell_j = U_1 + U_2$

$\vdots$ $\vdots$ $\vdots$

$k = 1$ $S_1 = nbm-p$ $V_1 = \sum_{j=1}^{nbm-p-1} \ell_j = U_1 + U_2 + \dots + U_{\beta-2} + U_{\beta-1}$

On a donc :

$$\begin{aligned} \sum_{k=1}^{\beta} V_k &= (\beta-1) U_1 + (\beta-2) U_2 + \dots + 2 U_{\beta-2} + U_{\beta-1} \\ &= \sum_{k=1}^{\beta} k U_{\beta-k} \quad \text{avec } U_0 = 0 \end{aligned}$$

et la formule (2) devient :

$$C = \beta (n+p-1) - \sum_{k=1}^{\beta} k U_{\beta-k}$$

$$C = \left(\frac{nbm-1}{p} \right) (n+p-1) - \sum_{k=1}^{\beta = \frac{nbm-1}{p}} k U_{\beta-k} \quad (3)$$

5.4.2. Discussion

4.2.1. : Cas où $p = nbm-1$ (il y a une seule tranche $\beta=1$).

D'après (3) on a $C = n + p - 1$ qui est indépendant des longueurs

des monotones.

Cas particulier : Si $nbm=n$ c.à.d. chaque processeur range un seul élément on a $C = 2 (n-1)$.

Donc cet algorithme est en $O(n)$ en parallèle.

4.2.2. Cas où les monotonies ont une même longueur ℓ ($n = \ell nbm$)

D'après la définition de U_k on a $U_k = p\ell$ pour tout k compris entre 1 et β .

$$\text{et } \sum_{k=1}^{\beta} k U_{\beta-k} = \sum_{k=1}^{\beta-1} p\ell k \quad \text{car } U_0=0$$

$$= \frac{1}{2} p \ell \beta (\beta-1) = \frac{1}{2} p \ell \left(\frac{nbm-1}{p} \right) \left(\frac{nbm-1}{p} - 1 \right)$$

$$\text{et } C = \frac{1}{2} \left(\frac{nbm-1}{p} \right) \left(2 (n + p - 1) - \ell (nbm - p - 1) \right)$$

$$= \frac{1}{2} \left(\frac{nbm-1}{p} \right) (2n + (2 + \ell) p - \ell nbm + \ell - 2)$$

$$\text{or } n = \ell nbm \quad \text{donc } C = \frac{1}{2} \left(\frac{n-\ell}{\ell p} \right) (n + (2+\ell) p + \ell - 2)$$

Cas particulier : Si le tableau était trié ($\ell=1$), la formule précédente deviendrait

$$C = \frac{1}{2} \left(\frac{n-1}{p} \right) (n + 3p-1)$$

- Si on dispose d'un seul processeur ($p=1$) on aura :

$$C = \frac{(n-1)(n+2)}{2}$$

donc cet algorithme, avec un seul processeur, est de $O(n^2)$.

5.5 Application de cet algorithme au tri par sélection quadratique

5.5.1. Principe séquentiel de la sélection quadratique (LOUI [25])

On partitionne le tableau à trier en R sous-tableaux de q éléments chacun au plus. Une première étape du tri consiste à chercher le plus petit de tous les sous-tableaux et de le mettre dans une case correspondante dans un fichier annexe (à R élément) ; on cherche, ensuite, le plus petit de ce dernier dont on repère l'indice par k , et on le place dans un fichier résultat. On reprend la recherche du plus petit dans le $k^{\text{ième}}$ sous-tableau et on le met dans la case vidée auparavant. On continue ce procédé jusqu'à ce que tous les sous-tableaux soient vides.

Le coût de cette méthode séquentielle est de Rq ($R+q$) opérations, il est optimum pour une partition équilibrée c.à.d. $R = q \simeq \sqrt{n}$, donc l'algorithme est de $O(n\sqrt{n})$ (n est le nombre d'éléments du tableau à trier).

5.5.2. Principe de la parallélisation

Le partitionnement du tableau à trier est le même qu'en séquentiel ; la parallélisation consiste en 2 étapes :

étape 1 : Activer R processeurs, en même temps, sur les R sous tableaux. Chaque processeur fait un tri autonome en choisissant un bon algorithme séquentiel. Le coût de cette étape est en $O(q \log_2 q)$.

étape 2 : On dispose de R monotonies triées, une fois terminé l'étape précédente, il suffit donc d'appliquer l'algorithme "Interclassement parallèle" énoncé en 5.3., en lançant $R-1$ processeurs.

D'après 5.4.2.1., cette étape se fait en $n + R - 2$ opérations.

Le coût global de cette méthode en parallèle est de $n + q \log_2 q + R - 2$ opérations.

5.5.3. Choix de q et R :

Cela revient à optimiser la quantité $C(q, R) = n + q \log_2 + R - 2$

avec la contrainte $G(q, R) = Rq - n \geq 0$.

Par application des multiplicateurs de LAGRANGE, on obtient :

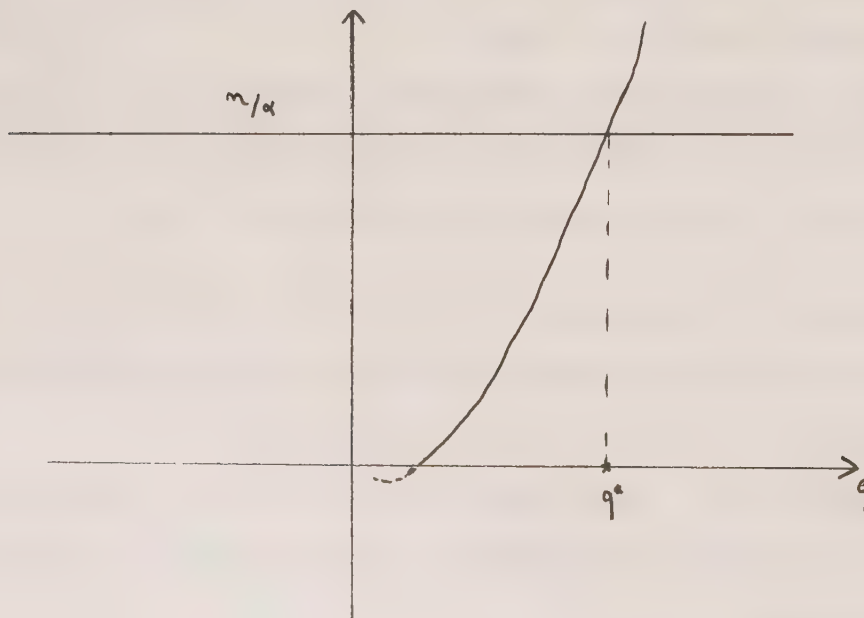
$$\left. \begin{aligned} \frac{\partial C}{\partial q} + \lambda \frac{\partial G}{\partial q} &= 0 \\ \frac{\partial G}{\partial R} + \lambda \frac{\partial G}{\partial R} &= 0 \end{aligned} \right\} \Rightarrow \begin{cases} \alpha (\log q + 1) + \lambda R = 0 & (1) \text{ avec } \alpha = \frac{1}{\log 2} \\ \lambda q + 1 = 0 & (2) \end{cases}$$

On multiplie (1) par $(-q)$ et (2) par (R) et on additionne. On obtient :

$$R - \alpha q (\log q + 1) = 0$$

$$\text{pour } Rq = n \Rightarrow \frac{n}{q} = \alpha q (\log q + 1)$$

$\frac{n}{\alpha} = q^2 (\log q + 1)$; graphiquement, on a une solution unique ; en effet si on trace les graphes des deux fonctions $\frac{n}{\alpha}$ et $q^2 (\log q + 1)$, on obtient un point d'intersection qui est la solution voulue.



5.6. Tri par comptage en parallèle

5.6.1. Principe séquentiel

Soit A un tableau de n éléments $(A_i)_{i=1,n}$ à trier. On associe à chaque élément A_i un compteur C_i (initialisé à 0) pour compter le nombre d'éléments qui lui sont inférieurs, en le comparant à tous ses prédécesseurs (s'il en a). Après avoir traité tous les éléments du tableau A, on range chaque A_i dans un fichier auxiliaire B à l'indice C_i+1 et on range ensuite ce fichier résultant dans le tableau A.

Algorithme séquentiel :

début

pour $i := 2$ à n faire

pour $j := 1$ à $i-1$ faire

si $A(i) < A(j)$ alors $C(j) := C(j) + 1$

sinon $C(i) := C(i) + 1$;

pour $i := 1$ à n faire

$B[C(i) + 1] := A(i)$;

pour $i := 1$ à n faire

$A(i) := B(i)$;

fin.

5.6.2. "Parallélisation" de cet algorithme

Nous pouvons envisager une première solution qui consiste à activer $n-1$ processus $Q_1, Q_2, \dots, Q_{n-1}$, simultanément de la façon suivante : chaque processus Q_i traite l'élément A_{i+1} en le comparant à A_i , puis à $A_{i-1}, \dots, A_1$ pour tout i ($1 \leq i \leq n-1$). Cela impose à protéger les compteurs $(C_j)_{j=1,n}$; par exemple Q_i , en comparant A_{i+1} à A_i , pourrait chevaucher sur C_{i+1} avec Q_{i+1} qui compare A_{i+2} à A_{i+1} . Pour remédier à cela, nous pouvons utiliser le "parallélisme "à la chaîne" avec la contrainte suivante :

Q_i ne peut avancer que s'il compte, à tout instant, au moins deux comparaisons de moins par rapport à Q_{i+1} pour tout i ($1 \leq i \leq n-1$) (type DE MASSAS [26]).

Pour éviter les chevauchements et les attentes, nous retiendrons la seconde solution qui consiste à activer n processus en parallèle (au lieu de $n-1$).

Chaque processus Q_i compare l'élément A_i à tous les autres éléments du tableau A et enregistre, dans une variable locale C , le nombre d'éléments qui lui sont inférieurs. Après avoir fini ce traitement, on range A_i dans un tampon B à l'indice $C+1$. Ceci est indépendant du travail des autres processus.

Algorithme parallèle :

pour $i := 1$ à n en parallèle faire

(* i est l'index du processus Q_i *)

début

$C_p \leftarrow \text{Cardinal } \{ A(j) / A(i) \geq A(j), \text{ pour } j < i \}$ (1)

$C_s \leftarrow \text{Cardinal } \{ A(j) / A(i) > A(j), \text{ pour } j > i \}$ (2)

$C = C_p + C_s$; $B(C+1) \leftarrow A(i)$

fin

Commentaires et remarques :

- (1) indique qu'on compare $A(i)$ à ses prédécesseurs
- (2) indique qu'on compare $A(i)$ à ses successeurs
- Les différents compteurs (C) des processus sont distincts à la fin de leurs exécutions.
- Dans chaque processus Q_i , on ne tient compte des éléments égaux à A_i que parmi les prédécesseurs de ce dernier, et non pas parmi ses successeurs ...

Sinon, on trouverait égalité des compteurs pour les éléments égaux du tableau A, cela poserait un problème pour les ranger.

5.6.3. Exemple : soit à trier le tableau suivant :

$$A = \begin{array}{|c|c|c|c|c|c|} \hline 4 & 2 & 1 & 3 & 1 & 2 \\ \hline 1 & 2 & 3 & 4 & 5 & 6 \\ \hline \end{array}$$

Processus Q_1 ($I = 1$, on cherche à placer $A(1)=4$)

prédécesseurs de 4 = $\emptyset$ $C_p=0$

successeurs de 4, inférieurs à 4 : $\{2, 1, 3, 2\}$ donc $C_s=5$ et par

conséquent $C=5$

rangement de $A(1)=4$, dans B, à l'indice 6

$$B \begin{array}{|c|c|c|c|c|c|} \hline & & & & & 4 \\ \hline 1 & 2 & 3 & 4 & 5 & 6 \\ \hline \end{array}$$

Processus Q_2 ($I = 2$, $A(2) = 2$)

prédécesseurs de 2, $\leq 2 = \emptyset \Rightarrow C_p = 0$

successeurs $< A(2) = \{1, 1\} \Rightarrow C_s = 2$ et donc $C=2$

$$B \begin{array}{|c|c|c|c|c|c|} \hline & & 2 & & & \\ \hline 1 & 2 & 3 & 4 & 5 & 6 \\ \hline \end{array}$$

Processus Q_3 ($I = 3$, $A(3) = 1$)

prédécesseurs $\leq A(3) = \emptyset$ $C_p=0$

successeurs $< A(3) = \emptyset$ $C_s=0$ et donc $C=0$

$$B \begin{array}{|c|c|c|c|c|c|} \hline 1 & & & & & \\ \hline 1 & 2 & 3 & 4 & 5 & 6 \\ \hline \end{array}$$

Processus Q_4 ($I = 4$, $A(4) = 3$)

prédécesseurs $\leq A(4) = \{2, 1\}$ $C_p=2$

successeurs $< A(4) = \{1, 2\}$ $C_s=2$ et donc $C=4$

$$\begin{array}{|c|c|c|c|c|c|} \hline & & & & 3 & \\ \hline 1 & 2 & 3 & 4 & 5 & 6 \\ \hline \end{array}$$

Processus Q_5 ($I = 5$, $A(5) = 1$)

Prédécesseurs $\leq A(5) = \{1\}$ $C_p = 1$

Successeurs $< A(5) = \emptyset$ $C_s = 0$ donc $C = 1$

	1				
1	2	3	4	5	6

Processus Q_6 ($I = 6$, $A(6) = 2$)

Prédécesseurs $\leq A(6) = \{2, 1, 1\}$ $C_p = 3$

Successeurs $< A(6) = \emptyset$ $C_s = 0$ et donc $C = 3$

			2		
1	2	3	4	5	6

Conclusion : A la fin de l'exécution de tous les processus, le tableau B contient tous les éléments de A triés dans l'ordre croissant

B	1	1	2	2	3	4
	1	2	3	4	5	6

5.6.4. Programmation parallèle de l'algorithme

Début : A (1:N) est le tableau à trier

B (1:N) est un tableau intermédiaire

Ces deux tableaux sont communs à tous les processus.

Pour I:= 1 Haut N EN PARALLELE FAIRE

Début du Bloc II

(*chaque processus possède en propre les variables suivantes :

I, J et C*)

C:= 0 ; (*initialisation du compteur à 0*)

J:= 1 ;

TANT QUE (J < I) ET (J ≤ N) FAIRE

début

SI A[I] ≥ A[J] ALORS C:= C+1 ;

J:= J+1

Fin ;

J:= J+1 ;

TANT QUE (J > I) ET (J ≤ N) FAIRE

début

SI A[I] > A[J] ALORS C:= C+1 ;

J:= J+1

fin ;

B[C+1] := A[I]

Fin du bloc II

Pour I:= 1 HAUT N FAIRE A[I] := B[I] ;

Fin

L'instruction parallèle est du type "avec" ; elle pourrait être implémentée par la primitive NFØRK qui assure la gestion du parallélisme et lance les processus non encore exécutés dans le cas où le nombre de processeurs disponibles est inférieur

à N. Le dernier processeur qui place A_N , continue en séquence pour ranger le tableau B dans le tableau A (dernière instruction du programme).

5.6.5. Complexité

Soit p le nombre de processeurs disponibles, le nombre d'opérations demandé par cet algorithme, est alors égal à $\frac{n(n-1)}{p}$.

En effet, dans chaque processus, il y a $n-1$ comparaisons.

Supposons que n est multiple de p , on aura donc le nombre total de comparaisons

"nc" qui est :

$$nc = \sum_{k=1}^{n/p} n-1 \Rightarrow nc = \frac{n(n-1)}{p}$$

Si $p = n$ l'algorithme est alors en $O(n)$

Si $p = 1$ l'algorithme est alors en $O(n^2)$.

5.7. Autres algorithmes de tri parallèle (TRE[37])

5.7.1. Sélection (DEMAS[26])

On utilise p processeurs pour trier un fichier de n articles, qui cherchent l'élément maximum entre les indices $[1, n]$, puis $[1, n-1]$, ..., $[1, n-p+1]$, Ces processeurs sont décalés d'une opération.

5.7.2. Quick sort (RACH[23], TR[37]).

L'idée est de faire un partitionnement du tableau à trier en p sous tableaux équilibrés et d'affecter chaque sous-tableau à un processeur qui fera un tri local par la méthode du Quick sort.

Le coût global de cette méthode dans le cas le plus favorable est de $3n + 1,4 \left(\frac{n}{p} - 1\right) \log \left(\frac{n}{p} - 1\right)$ opérations ($p < n$)

5.7.3. Tri externe "par interclassement" : (S.EVEN[13])

On utilise p processeurs et $4p$ fichiers, le tri se fait en 3 étapes :

- 1°) on sépare le fichier du départ en $2p$ fichiers,
- 2°) on interclasse chaque groupe de 2 fichiers sur p processeurs indépendamment,
- 3°) on interclasse les p fichiers triés obtenus.

L'algorithme coûte sensiblement $N/p \log_2 N + 3N$.

BIBLIOGRAPHIE

- [1] ATKINSON R.R., HEWITT C.E.-
"Specification and Proof Techniques for Serializers"
IEEE Trans. on Software Engineering.
Vol. SE-5, n° 1, Janv. 1979

- [2] BATCHER K.E.-
"Sorting networks and their applications"
Proc. AFIPS 1968 SICC, vol 32, AFIPS Press, Montvale, N.J.
pp. 307-314

- [3] BERNSTEIN A.J.-
"Analysis of programs for parallel processing"
IEEE Trans. on Computers, vol 15, n° 5, Oct 1966

- [4] BOUKNIGHT W.J., STEWART A.-
"The ILLIAC IV System"
dans "le traitement parallèle",
Ecole de l'IRIA du 29 mai au 2 juin 1978 (volume 2)

- [5] BRINCH HANSEN P.-
"Structured multiprogramming"
Comm. ACM 15,7 (1972)

- [6] BRINCH HANSEN P.-
"A comparison of two synchronizing concept"
Acta Informatica 1,3 (1972)

- [7] CAMPBELL R.H., HABERMAN A.N.-
"The specification of process synchronization by path expression"
dans "le traitement parallèle"
Ecole de l'IRIA du 29 mai au 2 juin 1978 (volume 1)

- [8] COMPTE D., SYRE J.C.-
Le système LAU
Rapport de recherche CERT, Toulouse 1976

- [9] CONWAY M.E.-
"A multiprocesseur system design"
Proceedings AFIPS, Fjcc 1963

- [10] COURTOIS P.J., HEYMANS F., PARMAS D.-
"Concurent Control with READERS and WRITERS"
Comm. ACM, 14, 10 (1971)

- [11] CROCUS
Systèmes d'exploitation des ordinateurs
Dunod.

- [12] DIJKSTRA E.W. -
"Cooperating sequential processes"
Programming langages. Academic Press 1968.

- [13] EVEN S.-
"Parallelism in tape-sorting"
Comm. ACM, 17, 4 (1974)

- [14] FLYNN M.J.-
"Some Computer organizations and their effectiveness"
IEEE Trans. on Electronic Computers, Vol C-21, n° 9, 1972

- [15] GIRAULT C.-
Réseaux de Pétri et synchronisation de processus
Institut de programmation, CNRS-ERA 592, PARIS.
- [16] GONZALEZ M.J., RAMAMOORTHY C.V.-
"A survey of techniques for recognizing parallel processable streams
in computer programs"
AFIPS, FJCC 1969.
- [17] HOARE C.A.R.-
"Towards a theory of parallel programming"
Operating system techniques. Academic Press, London 1972.
- [18] HOARE C.A.R. -
"Monitors : an operating system structuring concept"
Comm. ACM 17, 10 (1974)
- [19] HOARE C.A.R. -
"An axiomatic basis for computer programming"
Comm. ACM; Vol 12 (1969)
- [20] KAISER C.-
Conception et réalisation de systèmes à accès multiple : gestion
du parallélisme
Thèse d'état, PARIS IV, 1973
- [21] KESSELS J.L.W.-
"An alternative to even queues for synchronization in monitors"
Comm. ACM, 20, 7, 1977
- [22] KNUTH D.E.-
"The art of computer programming : sorting and searching"
T.3, Addison Wesley, 1973

- [23] KUNG H.T., THOMPSON C.D.-
"Sorting on a Mesh-connected parallel computer"
Comm. ACM, 20, 4 (1977)
- [24] LATEUX M.-
Synchronisation de processus
Publication n° 16 du Laboratoire de calcul de Lille. Mai 1978.
- [25] LOUIT G.-
Algorithmes de tri
Monographies d'informatique AFCET, n° 9, Dunod 1971
- [26] DE MASSAS E.-
Algorithme de tri parallèle en pastoral
Bigre, n° 9, Avril 1978
- [27] MAZARE G.-
Structures multiprocesseurs, problèmes de parallélisme, définition et
évaluation d'un système particulier.
Thèse d'état, Grenoble, 1978.
- [28] MOALLA M., PULOU J., SIFAKIS J.-
Réseaux de Petri synchronisés
RAIRO. Automatique, vol 12, n° 2, 1978
- [29] NAKECHBANDI M.-
Quelques problèmes numériques et non numériques en parallèle
Thèse de 3e cycle, Besançon, 1979
- [30] OWICKI S., GRIE D.-
"An axiomatic proof technique for parallel programmes"
Acta Informatica, vol 6, 1976.

- [31] PERIN G.-
Ecriture et vérification de programmes parallèles
4ème colloque international sur la programmation, PARIS, 22-24 avril 1980.
- [32] PERIN G.-
"Outils de conception de programmes parallèles cadencés par les données"
Bigre n° 14, Avril 1979
- [33] RACHIDI A.R. -
Algorithme de tri en parallèle : Quicksort
DEA Nancy, Juin 1978
- [34] ROBERT P., VERJUS J.P.-
Expression autonome de la synchronisation de processus concurrents.
Journées de formation : Programmation globale des synchronisations
en temps réel, PARIS, Nov. 1977
- [35] ROBERT P., VERJUS J.P.-
"Towards autonomous descriptions of synchronization modules"
Proceeding of IFIP 1977 Congress, north Holland Pub. CO,
Amsterdam (1977)
- [36] TREHEL M.-
Multiprocesseurs, algorithmes, architectures, langages, systèmes.
Notes de cours de DEA, 1979, Besançon
- [37] TREHEL M. -
Etude du Quicksort en parallèle.
Ecole d'été d'informatique de l'AFCEt, NAMUR, Juillet 1978
- [38] TREHEL M. -
Typologie d'algorithmes parallèles.
Ecole d'été d'informatique de l'AFCEt, Aix en Provence, Juillet 1980.

